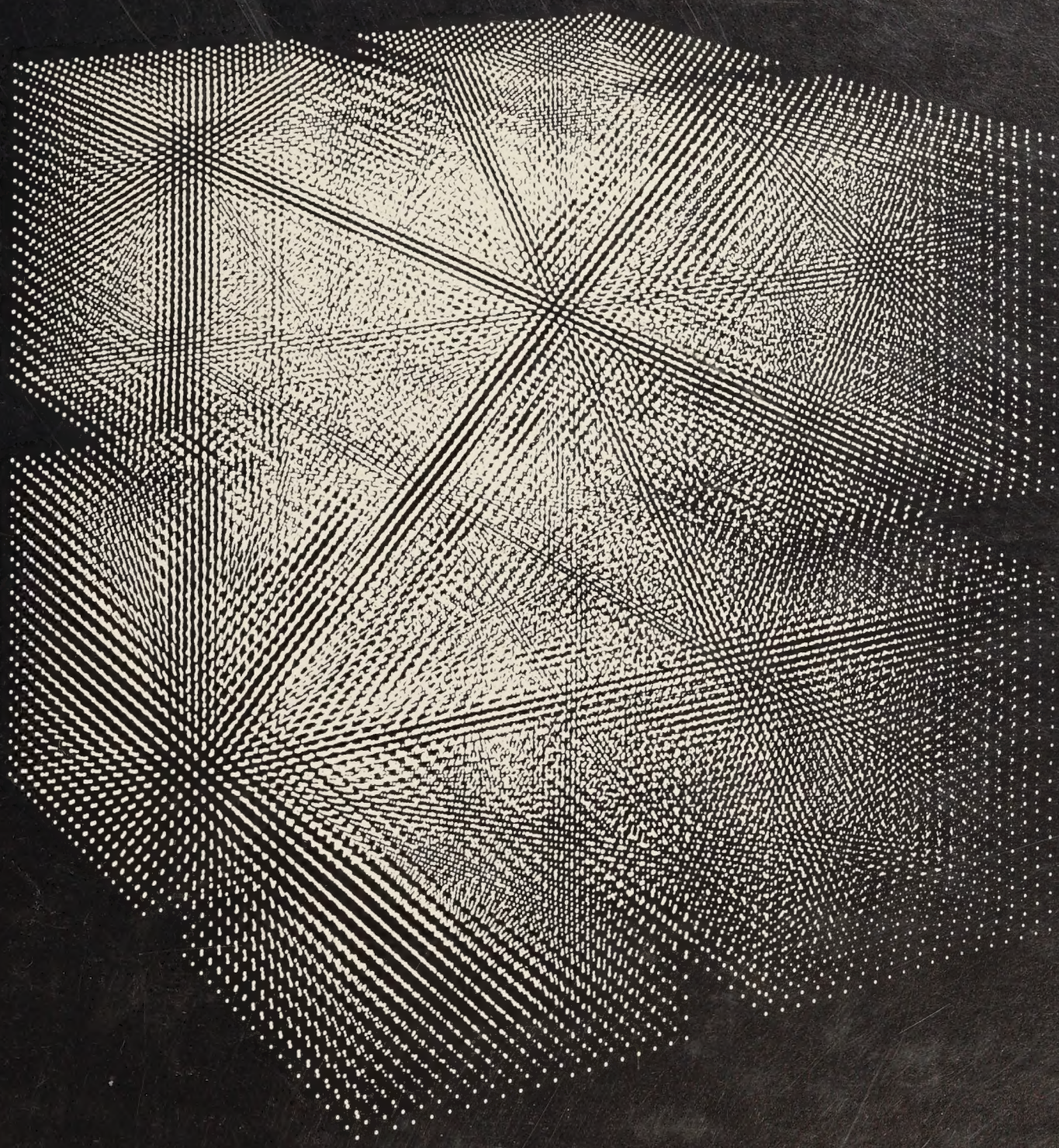
Thinking Machines Corporation

The Connection Machine® System



Guide to Documentation

Documentation Set Contents

Volume I
Guide to Documentation
Connection Machine User's Guide: Using the Symbolics 3600 as a Front End
*The Essential *Lisp Manual*
Connection Machine Parallel Instruction Set (Paris): The Lisp Implementation
User Contributed Software
Release Notes

Volume II
Guide to Documentation
Connection Machine User's Guide: Using a Unix System Front End
Connection Machine Parallel Instruction Set (Paris): The C Implementation
Introduction to Data Level Parallelism

Overview

The languages used to program the Connection Machine are *LISP, the LISP implementation of PARIS, and the C implementation of PARIS. The table below shows the programming environments and front ends to the Connection Machine from which one programs. The diagram on the facing page and "What To Read" below show the manuals relating to the particular languages and environments.

Language	Description, Programming Environment
*LISP	Based on the LISP language Used within a LISP environment
PARIS	Parallel Instruction Set The Connection Machine's assembly language
PARIS (LISP):	Used within a LISP environment
PARIS (C):	Used on a UNIX system

Programming Environment	Front End on Which Environment Is Currently Available
LISP	Symbolics LISP Machine Lucid LISP on a UNIX operating system
UNIX	Digital Equipment Corporation VAX front end, running the ULTRIX operating system (ULTRIX is Digital's implementation of UNIX)

What To Read

Of Interest to All Connection Machine Users

Begin with the User's Guide appropriate for you, and continue with:
Release Notes (Volume I)
Introduction to Data Level Parallelism (Volume II)

Applies to Either LISP Environment

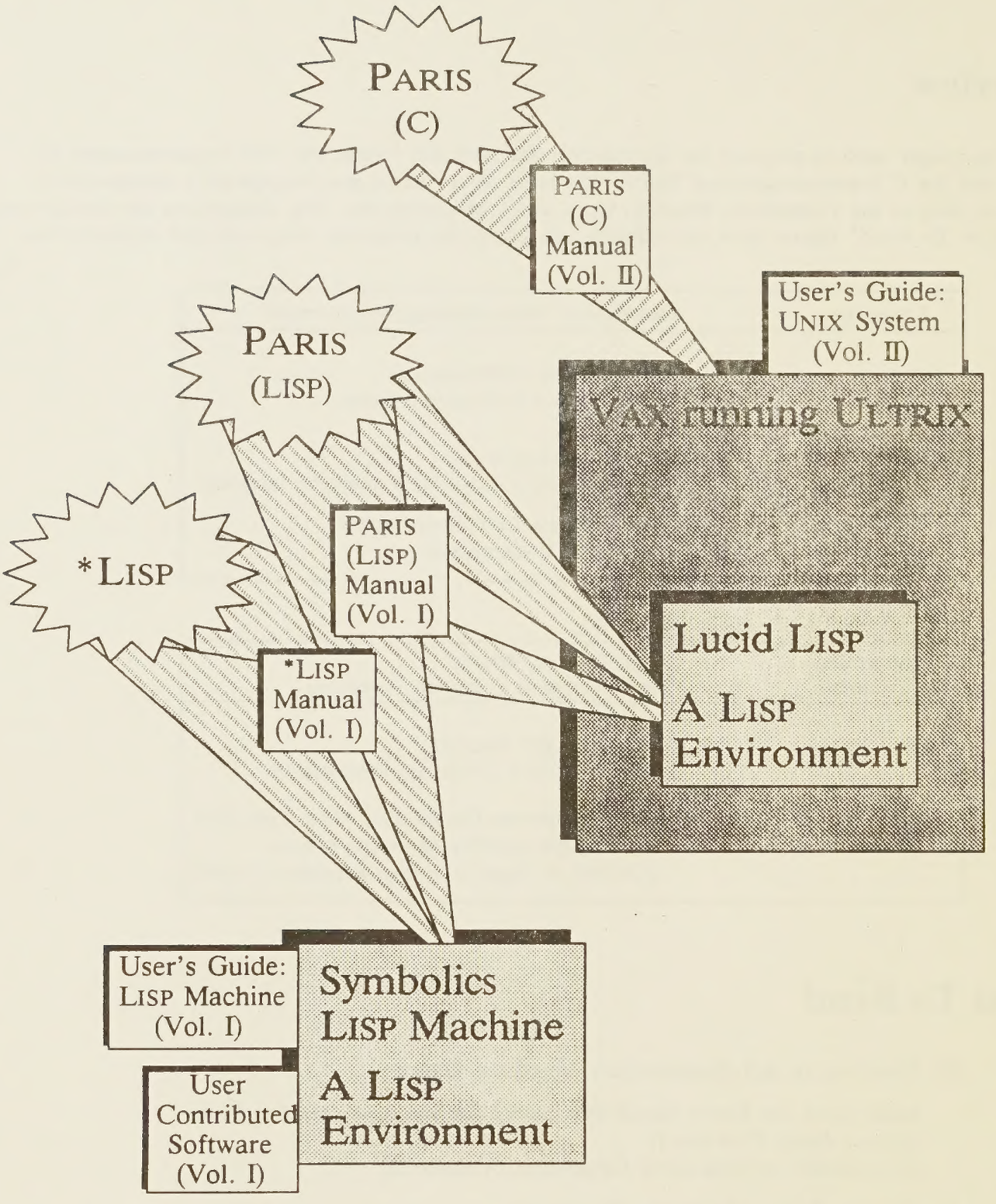
*The Essential *LISP Manual* (Volume I)
Parallel Instruction Set (PARIS): The LISP Implementation (Volume I)

Specific to a LISP Machine Front End

User's Guide: Using the Symbolics 3600 As a Front End (Volume I)
User Contributed Software (Volume I)

Specific to a UNIX System Front End

User's Guide: Using a UNIX System Front End (Volume II)
Parallel Instruction Set (PARIS): The C Implementation (Volume II)



Languages, programming environments, front ends, and manuals for Connection Machine programming, as described on facing page.

Connection Machine User's Guide

Using a UNIX System Front End

Release 4.0A Field Test

DRAFT

Thinking Machines Corporation
Cambridge, Massachusetts

First printing, April 1987

Thinking Machines Corporation
245 First Street
Cambridge, MA 02142-1214
(617) 876-1111

Contents

Learning Connection Machine Programming

Learning something new can be difficult or exciting, depending on your attitude toward the new skill, whether you feel you are given the tools you need to learn it, and of course whether the new skill seems exciting enough to be worthwhile. Either way, learning a new skill requires energy, whether it is energy to overcome difficulties or energy to fuel the interest as you move from one step to the next.

The new skill here is programming the Connection Machine: learning to see the natural parallel divisions of a problem, learning to devise algorithms that take advantage of these parallel divisions, and learning to use the languages available to write and run programs that make the thousands of Connection Machine processors work on the problem. In short, this is learning a new way to solve problems that is inherently faster, and hence interesting, because it requires making many—thousands—of operations happen at once, at the data level.

Guide To Tools

Although there is nothing inherently difficult about writing programs for a data level parallel computer, few have experience with it. The ability to see the natural parallel divisions in a problem comes with experience. An excellent tool for this learning step is *Introduction to Data Level Parallelism*, in Volume II of The Connection Machine Documentation Set. It contains several case studies of applications programs written for the Connection Machine system and is a rich source book for someone new to data level parallelism.

For those using a UNIX operating system on the front end to the Connection Machine, the programming languages used are C and PARIS. PARIS stands for "Parallel Instruction Set;" its instructions are executed on the Connection Machine and are analagous to a serial machine's assembly language instructions. The tool for learning it is *Connection Machine Parallel Instruction Set (PARIS): The C Implementation* in Volume II. Those who need to learn the C language should obtain *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie. *Introducing the UNIX System* by Henry McGilton and Rachel Morgan is a good book for those learning a UNIX operating system.

Now we come to the purpose of this user's guide. Programs written on a front end running a UNIX system, currently a Digitial Equipment Corporation VAX running ULTRIX 2.0, are written in C with function calls to PARIS; PARIS is implemented as a library of C functions. This guide is the tool for learning how to run such C/PARIS programs on a UNIX front end, with the PARIS calls executed by the Connection Machine. It shows how the front ends communicate with the Connection Machine and how to attach and initialize part or all of the Connection Machine for your program's use. A short section explains powering on the Connection Machine and running hardware diagnostics.

Another user interface to the Connection Machine besides the UNIX operating system is a LISP environment. Usually, users preferring only the LISP environment have a Symbolics LISP machine as a Connection Machine front end, but UNIX users do have the option of invoking a LISP environment on their UNIX system, and programming the Connection Machine from there. The languages used are *LISP, which is built on top of COMMON LISP, and the LISP implementation of PARIS. Unlike C and C/PARIS under UNIX, *LISP and LISP/PARIS in a LISP environment are used as independent languages. Those interested in using a LISP interface to the Connection Machine from UNIX should read this manual first to learn how to invoke a LISP environment under UNIX, and should then turn to *The Essential *LISP Manual* and *Connection Machine Parallel Instruction Set (PARIS): The LISP Implementation* in Volume I to learn those languages.

For those interested in learning more about the design issues and philosophies that led to building the Connection Machine, *The Connection Machine* by W. Daniel Hillis is excellent reading.

Guide to Tools Summary

For learning to write data level parallel programs:

Introduction to Data Level Parallelism
Volume II of The Connection Machine Documentation Set

For learning a UNIX operating system:

Introducing the UNIX System by Henry McGilton and Rachel Morgan

For learning the C programming language:

The C Programming Language by Brian W. Kernighan and Dennis M. Ritchie

For learning C/PARIS:

Connection Machine Parallel Instruction Set (PARIS): The C Implementation
Volume II of The Connection Machine Documentation Set

For learning how to run C/PARIS programs on a UNIX front end:

Connection Machine User's Guide: Using a UNIX System Front End
Volume II of The Connection Machine Documentation Set

For learning how to use a LISP environment from UNIX:

Connection Machine User's Guide: Using a UNIX System Front End
Volume II of The Connection Machine Documentation Set

For learning how to program the Connection Machine with a LISP environment:

Connection Machine User's Guide: Using a Symbolics 3600 As a Front End
*The Essential *LISP Manual*
Connection Machine Parallel Instruction Set (PARIS): The LISP Implementation
Volume II of The Connection Machine Documentation Set

For learning about Connection Machine design philosophy:

The Connection Machine by W. Daniel Hillis

User Support

Of course, another learning tool comes in the form of people of whom you can ask questions. There are two ways to reach Thinking Machines Customer Support:

Electronic Mail
If your site can send electronic mail across the Arpanet, questions can be sent to the address "customer-support@think.com" (do not include quotes). This mailing list is closely monitored. You will receive a reply within 48 hours Monday through Friday; it will be either the answer to your question or an indication of when we will be able to send the answer.

Telephone
Call Thinking Machines at (617) 876-1111 and ask for Customer Support.

Notation Conventions

The notation conventions used in this guide are similar to those used in most UNIX documentation. Code fragments appear separate from text in a `typewriter` style font. User command names, names of files, and names of functions appearing in-line in text are also in a `typewriter` style font. Where a command and its syntax are being explained, the command is separate from the text in **boldface** type and the explanation appears below it. Optional arguments to commands appear in square brackets; the brackets are not part of the command syntax. Where the user is being instructed on what to type, what to type appears in **boldface**.

The Connection Machine System

A single Connection Machine system can be shared among up to four front-end users. At any given instance, the entire set of Connection Machine processors may be used by a single user, or the processors may be divided up among the four.

Within a single Connection Machine system, the processors are divided into portions of fixed size; each portion has either 8,192 or 16,384 processors. A single Connection Machine system can have one, two, or four 8192-processor portions, four providing a 32,768-processor system, or it can have one, two, or four 16,384-processor portions, four providing a full 65,536-processor system. Each user can have one, two, or all four portions.

Associated with each Connection Machine portion is an independent microcontroller. The microcontroller is responsible for interpreting the macro-instructions sent to it by the front end. Associated with the front-end computer is a bus interface that contains two FIFO (first-in/first-out) queues. The IFIFO (input FIFO queue) buffers macro-instructions and data issued from the front end to the microcontroller, and the OFIFO (output FIFO queue) buffers data being returned to the front end for operations such as `CM_read_array_by_cube_addresses`, which transfers values from Connection Machine processors to an array in the front end.

The front-end computer bus interface is connected to the microcontrollers through a 4×4 crossbar switch called the *Nexus*. (See Figure 1.) Before a front-end computer can use a Connection Machine system, or part of it, it must logically attach itself to one or more microcontrollers by reconfiguring the Nexus.

The machines currently available as front ends are the Symbolics 36xx LISP machine and the Digital Equipment Corporation VAX running the ULTRIX operating system, which is Digi-

tal's implementation of a UNIX system. The exact configuration of a Connection Machine system and front end(s) depends on the needs of a particular site. For example, a site may have just one VAX front end, or it may have one VAX and up to three LISP machines, each attached to one of the four Nexus ports. The VAX and LISP machines would be able to communicate to one another via an Ethernet connection between them.

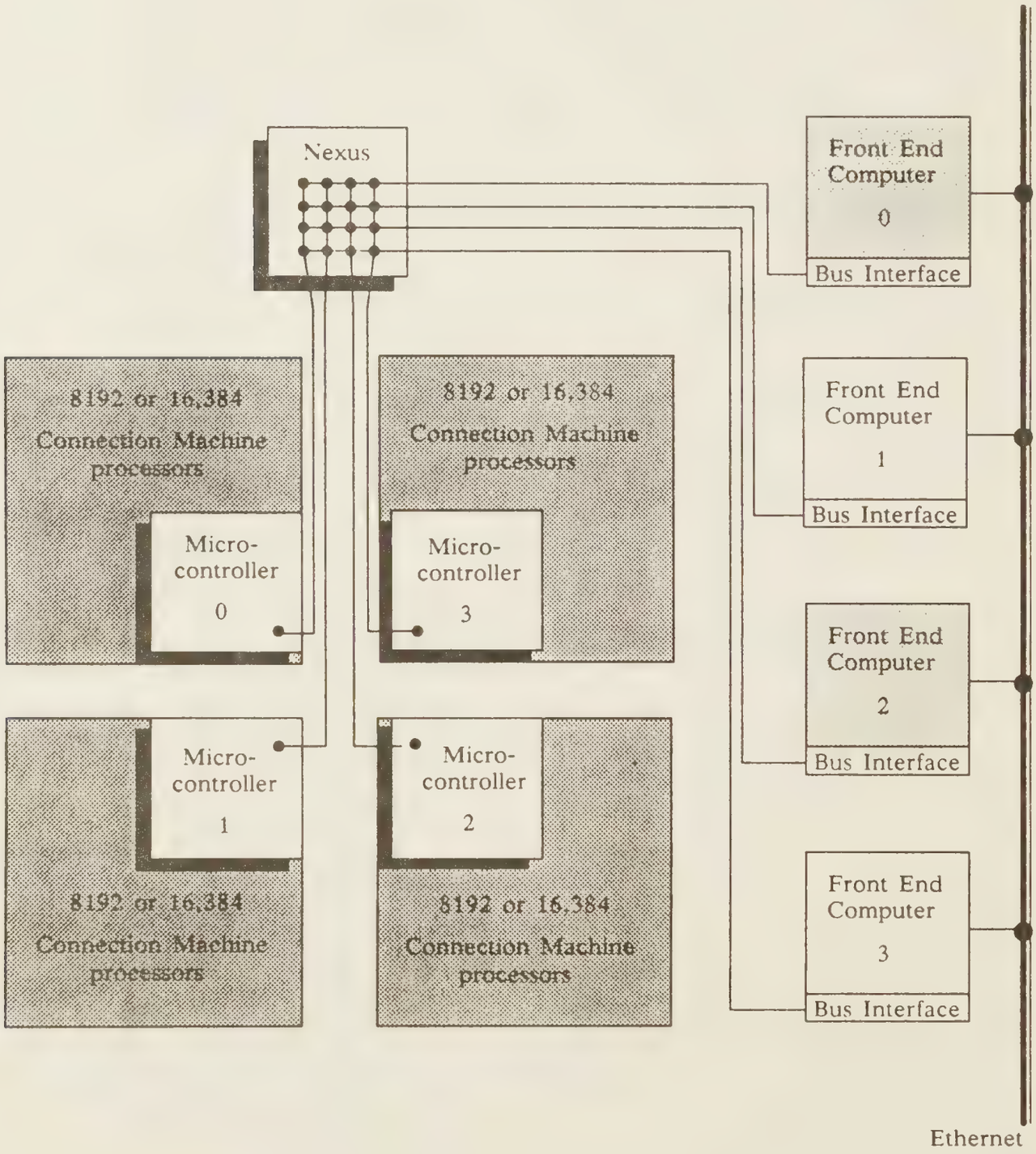


Figure 1. Connection Machine System: Microcontrollers, Nexus, and Front Ends

Using a UNIX System

There are three operations for allocating and initializing Connection Machine processors for use by a front-end computer running a UNIX operating system, two for finding out who else on the front end(s) is currently using the Connection Machine, and one for initially turning on power.

- **cmattach**
  Allocates hardware processors (and their associated microcontrollers) for use by a particular front end.
- **cmcoldboot**
  Completely resets the state of the Connection Machine hardware currently allocated to the front end, loads microcode into the control store of the microcontroller, initializes system tables, and clears user memory. Cold booting may be done to change the number of virtual processors between runs. However, it *must* be done between execution of different user programs.
- **cmdetach**
  Used to detach forcibly another user on the UNIX system, or another front end, from its Connection Machine processors. It can also be used to detach yourself.
- **cmusers**
  Lists who on the UNIX system is using the Connection Machine, their idle time, and users waiting to use the Connection Machine. This is similar to the UNIX who command.
- **cmfinger**
  Lists who on all the front ends is using the Connection Machine and what portions they are using.
- **cmpowerup**
  A once-only operation that initializes the Nexus, enabling access to the rest of the Connection Machine by the front ends. It is described in "For System Administrators."

The first task in programming the Connection Machine from a UNIX system is to write a C program with function calls to PARIS instructions, using your favorite UNIX editor. PARIS instructions are actually a library of C functions. The C/PARIS program is executed on the UNIX system front end, and the PARIS instructions imbedded in the C program are executed on the Connection Machine. *Global configuration variables* can also be used in C/PARIS programs. These variables contain information about the Connection Machine system that is needed in PARIS programming, and are set at the time of cold boot. (Configuration variables are listed and described in Chapter 3 of *Connection Machine Parallel Instruction Set (PARIS): The C Implementation*.)

The program must contain two particular lines of code, as shown in this C/PARIS program template:

```
#include <cm/cm.h>
main()
{
        CM_init()
        [C code]
        ...
        [PARIS function calls occurring within C code]
        [Global configuration variables used where needed]
        ...
}
```

The line `#include <cm/cm.h>` must occur at the beginning of the program. This statement sets up the C/PARIS programming environment by including the file defining the data types used by PARIS functions, such as `CM_memaddr_t` and `CM_cubeaddr_t`, and the global configuration variables.

The line `CM_init()` must occur in the program before any calls to PARIS functions are made. It "warm boots" the Connection Machine and initializes the values of the global configuration variables before the rest of the program is executed. This function has no arguments.

Warm booting the Connection Machine consists of clearing the error status indicators for the attached Connection Machine hardware, clearing the IFIFO and the OFIFO in the bus interface. The user memory areas in the Connection Machine system are not disturbed, but are checked for errors and any memory errors are reported. Certain system memory areas in the Connection Machine system are re-initialized, but the state of the pseudo-random number generator is not altered and the system lights display mode is not altered.

The next step is to compile the program. To be recognized by the C compiler, the name of the file containing the C/PARIS program must end with ".c". The `-l` option to the C compiler must be used when compiling a C/PARIS program, to make the library of PARIS functions available to the program. For example, the following compiles the C/PARIS program `life.c`:

```
cc life.c -lparis
```

The C compiler by default places the executable image in a file called `a.out`; normally, to run a C program you would simply type `a.out` to the UNIX prompt. However, there is a `-o` option that lets you specify the name of the executable file. For example, the following names it `life`:

```
cc life.c -lparis -o life
```

Typing **life** at this point would run the program, but before a C/PARIS program can be executed, Connection Machine processors must be allocated for its use. Attaching processors and running a C/PARIS program can be done in one of two ways, by batch access or interactive access, using the `cmattach` command.

cmattach [options] [program name] [program arguments]

This function allocates Connection Machine processors for use by the front end. The various options are described below. When `cmattach` is executed without options specifying the number of physical processors to attach, it attaches the minimum possible number of physical processors to the front end that issued the cmattach command. This is either 8192 or 16,384 processors, depending on whether the system is a 16k or 64k Connection Machine system. If no processors are available, cmattach prints a message to that effect and returns you to UNIX.

The default virtual processor configuration is one virtual processor per physical processor. After the processors are attached a cold boot is automatically performed (see below).

Batch Access

If a program name is specified on the cmattach command line it must be an executable C/PARIS program; in the example compilation above this would be `life`. The Connection Machine processors are attached and cold booted, and this program is run. Arguments to the C/PARIS program, if they exist, may be supplied on the command line. When the program is finished executing, the Connection Machine processors are automatically detached. For example, typing the following would attach the default number of processors, cold boot them, run the life program, and detach the processors (the percent sign is a typical UNIX prompt):

```
% cmattach life
```

Typically, a user may want to start up a C/PARIS program in the background. (Any UNIX command can be run in the background by following the command with the ampersand character, "&". The user is immediately returned to the UNIX prompt.) The following command causes 32k processors to be attached and cold booted with 64k virtual processors, at which point the C/PARIS program is executed. As specified by the -w option, if 32k processors are not free, cmattach waits until they become available. This example supposes that the program `life` takes one argument, whose value in this particular run is 27:

```
% cmattach -p 32k -v 65536 -w life 27 &
```

Interactive Access

The interactive facility allows you to have a session during which the front end remains attached to the Connection Machine, regardless of whether a C program is currently running.

If no C/PARIS program name is specified on the command line, cmattach attaches the requested number of physical and virtual processors, cold boots them, and places you in a subshell from which you can run the executable program by typing its name. Within this subshell, cmcoldboot can be used to change the number of virtual processors and cold boot them in one step. You can also change the number of physical processors and virtual processors by issuing a cmattach command, with the appropriate options, within the subshell. This will not create a new sub-subshell, but will simply attach and cold boot the requested number of processors. A UNIX subshell accepts any UNIX command, so within it you can also edit your program, re-compile it, and run it again, and you can issue the cmpowerup, cmusers, cmfinger, and cmdetach commands. The subshell is exited by typing **exit**, which automatically detaches your allocation of Connection Machine processors.

Cmcoldboot may be executed between C/PARIS program runs to reset the Connection Machine, but this might not be necessary—it depends on your application. Every time a program is executed within the subshell, a warm boot is perfomed (by CM_init) before the program is executed. This does everything that cold booting does except clear the processor's memory and re-load microcode. The currently selected set after a warm boot includes all processors. For some applications, not cold booting saves the time of re-loading data into the Connection Machine processors for every single program run when it is unnecessary. However, if a computation should be interrupted at some point and not continued, cmcoldboot should be exe-

cuted and the program should be started over from the beginning. If it is desired that the current run of a program not be affected by the state left from a previous one, `cmcoldboot` must be executed between program runs.

If `cmcoldboot` is executed without arguments, the same number of virtual processors is used as was set by the most recent `cmcoldboot` or `cmattach` command.

Arguments to cmattach

The optional arguments to `cmattach` allow you to modify its behavior, such as attach a different number of physical or virtual processors or prevent the automatic cold boot of the processors. Except where stated otherwise, any number of these options may be used at once; for example, `cmattach -p 8k -n -w` is appropriate.

-p physical-size
-u microcontroller

Either of these two optional arguments may be used to specify the number of physical processors desired, but not both at once. If neither argument is specified, the smallest possible amount of hardware will be allocated; this will be either 8,192 or 16,384 physical processors.

The *physical-size* argument must be one of the following:

8k or **8192**	Exactly 8,192 physical processors are to be allocated.
16k or **16384**	Exactly 16,384 physical processors are to be allocated.
32k or **32768**	Exactly 32,768 physical processors are to be allocated.
64k or **65536**	Exactly 65,536 physical processors are to be allocated.

The *microcontroller* argument must be one of the following:

0, **1**, **2**, or **3**	Exactly the specified microcontroller port is to be attached, regardless of whether that port controls 8,192 or 16,384 physical processors. (This option is useful primarily for hardware diagnostic procedures.)
0-1, **2-3**, or **0-3**	Exactly the specified microcontroller ports (0 and 1, 2 and 3, or all four) are to be attached, regardless of the number of physical processors involved. (This option is useful primarily for hardware diagnostic procedures.)

If the requested number of physical processors or set of microcontroller ports is not available, and the -w option has not been specified, a message is printed to that effect and you are returned to the operating system.

-v virtual-size
-x virtual-x -y virtual-y

Either the -v option alone, or the -x and -y options together, may be used to specify the number of virtual processors. The number of virtual processors is always greater than the number of physical processors. The virtual-size argument is an integer specifying the total number of virtual processors. The -x and -y options each require an integer; together they specify the size of the virtual NEWS grid of processors. This is a grid of nearest neighbors, the nearest neighbors being the processors to the North, East, West, and South (NEWS). For example, `cmattach -x 128 -y 512` specifies a virtual processor grid of 128 × 512 processors.

(See *Connection Machine Parallel Instruction Set (PARIS)* for more about virtual processors and the NEWS grid.)

-n

This option suppresses the automatic cold boot of the processors to be attached.

-w

This option causes cmattach to wait until the requested (or default) number of physical processors is available, if they are unavailable at the time they are requested, rather than return to the operating system.

Using cmcoldboot

cmcoldboot [dimension-0 [dimension-1]]

This operation completely resets the state of the hardware allocated to the executing front end, loads microcode, initializes system tables, and clears user memory.

The optional dimension arguments must be either a single integer or two integers. A single integer specifies the total number of *virtual* processors desired. Two integers specifies the desired size of the virtual NEWS grid. Each dimension must be a power of two, and must be greater than the default number for the physical configuration. The default for 8K processors is 64 × 128, for 16K processors it is 128 × 128, for 32K processors it is 128 × 256, and for 64K processors it is 128 × 512.

If the dimension arguments are not supplied, then the configuration of virtual processors defaults to those specified by the most recent `cmcoldboot` or `cmattach` operation preceding this one.

Bootstrapping a Connection Machine system includes the following actions:

- Loading microcode into the Connection Machine microcontroller and initiating microcontroller execution.
- Clearing and initializing the memory of allocated Connection Machine processors.
- Initializing all of the global configuration variables.
- Initializing the pseudo-random number generator by effectively invoking the operation `CM_initialize_random_number_generator` with no seed.
- Initializing the system lights display to a system supplied display mode by effectively invoking the operation `CM_set_system_leds_mode`. (This is not yet implemented on the VAX front end.)

If the cold-booting operation fails, then an error is signaled. If it succeeds, then three values are printed: the number of virtual processors, the number of physical processors, and the number of bits available for the user in each virtual processor. (These are exactly the values of the configuration variables `CM_user_cube_address_limit`, `CM_physical_cube_address_limit`, and `CM_user_memory_address_limit`.)

Obtaining Usage Information

The two commands `cmusers` and `cmfinger` can be issued from the UNIX operating system or from the interactive subshell to obtain information about the system.

cmusers

This is similar to the UNIX who command. It prints who on a UNIX system front end is using what portions of the Connection Machine system, who is waiting to use it, and idle time. A user who is attached but has been idle for a long time can be detached using cmdetach. Following is an example of cmuser's output:

```
% cmusers
Interface  User    Idle      Status

0          Rocky   0:02      ATTACHED          Running "cmattach"
           Moose   1:00      WAITING
%
```

cmfinger [name]

The cmfinger command prints a table indicating which front ends are connected to what portions of a given Connection Machine system. If the name of a front end is specified as an argument, information is reported for only that front end. If no name is specified (the argument is unsupplied), then information is reported for all front ends connected to the same Connection Machine system as the executing front end. For example:

```
% cmfinger
Connection Machine System Gemstone    Physical size: 64K processors, 4K RAM

Fred    Sapphire   Microcontroller Port (0 1) <-- 32768 physical processors
Wilma   Emerald    Not Attached to a Port
Betty   Ruby       Microcontroller Port (2)   <-- 16384 physical processors
Barney  Topaz      Microcontroller Port (3)   <-- 16384 physical processors
```

Here the Connection Machine system in question is named Gemstone, and it has 65,536 physical processors, each with 4,096 bits of memory. It is connected to four front ends named Sapphire, Emerald, Ruby, and Topaz, which are being used by users named Fred, Wilma, Betty, and Barney, respectively. Fred is using 32,786 physical processors; Betty and Barney are each using 16,384 processors; and Wilma is not using the Connection Machine system at all.

Detaching Idle Users

cmdetach [interface-number | front-end-name]

This command is used to detach from the Connection Machine users on the UNIX system who have been idle for a long time. To detach a user on the same front end as you, use cmusers to find the interface number of the person you want to detach, and use that number as the argument to cmdetach.

If a *front-end-name* argument is specified, it must be the name of a front end that is connected to the same Connection Machine system (that is, Nexus) as the front end executing the call. Specifying the name of some other front end forcibly detaches that other front end, possibly disrupting any ongoing interaction with the Connection Machine system. The external communications network is used to send a message to the detached front end to inform its user that it has been forcibly detached.

Debugging

For C/PARIS programs, the standard tools for debugging are used in conjunction with special user dependent routines to access the Connection Machine. The standard tool for debugging on UNIX operating systems is dbx, a source code level debugger that includes break points, stepping, tracing, etc. This section presents an example using dbx.

To use dbx you must first have compiled the C/PARIS program and created an executable image. In this example, the default name for the executable image, a.out, is used. To enter dbx, type:

```
% dbx a.out
(dbx)
```

The (dbx) is the dbx prompt. At this point a variety of dbx commands can be used to run the program and stop at break points, step through line by line, and more; see the ULTRIX manuals for more information about dbx. In addition to dbx commands, functions defined in your C/PARIS programs can be called while in dbx. The following is an example of such a routine, written for debugging. It is used to examine unsigned integers on the Connection Machine.

```
void db_read_u(addr, len, start, end, by)
unsigned int addr, len, start, end, by;
{
  unsigned long data;
  if (by < 1) by=1;
   printf("for address: %d, length: %d, processor start: %d,
           end: %d, by: %d\n", addr, len, start, end, by);
   printf("Proc\tData\n");
   for (;start <= end;start += by)
   {
     data = CM_u_read_from_processor(
            (CM_cubeaddr_t)start, (CM_memaddr_t)addr, (unsigned)len);
     printf("%d\t%D\n", start, data);
   }
}
```

This routine prints unsigned integers taken from a particular location in certain processors in the Connection Machine. For example, if dbx is running with a C/PARIS program that includes the routine db_read_u above, the following command can be entered to display the unsigned integer at location 100, of length 16, for processors 0, 10, 20, and 30:

```
(dbx) call db_read_u(100,16,0,30,10)
```

The output produced is:

```
for address: 100, length 16, processor start: 0, end: 30, by: 10
Proc            Data
0               255
10              35
20              65000
30              1734
```

Similar routines can be written to display strings, floating point numbers, and signed integers. Analogous routines for placing data into the Connection Machine can be written. If a program uses NEWS addresses instead of cube addresses, routines can be written to display and store data by NEWS address in addition to cube address.

Any PARIS instruction can also be called from within dbx in this way. However, to be known to dbx it must be included in the loaded program. To make PARIS functions available to dbx that would not otherwise be used in the C/PARIS program, simply include in the program a routine that is never actually called, and inside of it call the PARIS functions you need for debugging. The presence of the PARIS functions there will cause them to be loaded in when you enter dbx, even though they are never actually used by the C/PARIS program.

Errors

If there is an error, for example, "floating point exception, illegal divide by zero," the Connection Machine detects these, the front end prints them, and the program stops. You cannot write the C/PARIS program so that it tries to continue when this happens.

Using the LISP Environment

There are four possible LISP worlds, or saved images, one of which may be entered at a time by a particular user. To enter a LISP environment and write *LISP programs, to the UNIX prompt type:

```
% /usr/local/starlisp
```

To enter a LISP environment and write PARIS programs or run diagnostics, type:

```
% /usr/local/lisp-paris
```

To enter a LISP environment and use the *LISP simulator, type:

```
% /usr/local/starlisp-sim
```

To enter a LISP environment and user the PARIS simulator, type:

```
% /usr/local/lisp-paris-sim
```

Any one of the above four commands places you in a LISP environment containing the software appropriate for what you want to do. How to use the Connection Machine from within a LISP environment is described in *Connection Machine User's Guide: Using the Symbolics 3600 As a Front End*, *The Essential *LISP Manual*, and *Connection Machine Parallel Instruction Set (PARIS): The LISP Implementation* in Volume I of the Connection Machine Documentation Set. Using the LISP environment itself is explained in the Lucid LISP documentation.

To exit the LISP environment, type:

```
(system:quit)
```

For System Administrators

The Connection Machine software provided under UNIX consists of the following files:

—**libparis.a**
cm.h
A library of subroutines, and its associated include file, which is a complete C implementation of PARIS. All the PARIS instructions can be called from C language programs. This library includes routines for error handling and reporting.

—**cm.c**
cmreg.h
cmvar.h
cmioctl.h
The ULTRIX driver for controlling the Bus Interconnect Bus Interface (BIBI) and thus communicating with the Connection Machine, and its three associated include files.

—**/dev/cm***
The Connection Machine, as a device under the ULTRIX system. There is one device for each person using the Connection Machine.

Turning On Power

cmpowerup

This function resets the state of the Nexus, causing all front-end computers to become logically detached from the Connection Machine system. When a Connection Machine system is first turned on or is to be completely reset for other reasons, this is the first operation to perform. Any of the front-end computers may be used to do it. If users on other front-end computers are actively using the Connection Machine system, their computations will be disrupted. The person issuing the cmpowerup command will be asked, "There may be other CM users; they will be disrupted by powering up. Do you really want to do this? (y,n)," and thereby is given the chance to abort.

Basic Connection Machine Operation

Near the front center of the Connection Machine is an Emergency Power Off Switch. Pressing the Emergency Power Off Switch turns off power to the entire machine, and is intended for emergencies only. The switch trips several circuit breakers in the machine, and a Thinking Machines Field Engineer must be called to reset these breakers before the machine can be restarted.

Directly above the Emergency Power Off Switch on the front of the Connection Machine is a control panel which is used both for turning on and off the power to 16K units of the machine, and for displaying the status of connections between front end processors and the Connection Machine hardware. Figure 2 shows the control panel and emergency power switch. This figure shows the control panel used on the 16K processor systems. A later figure (Figure 3) shows the control panel for a 64K processor system; there are minor differences between the two.

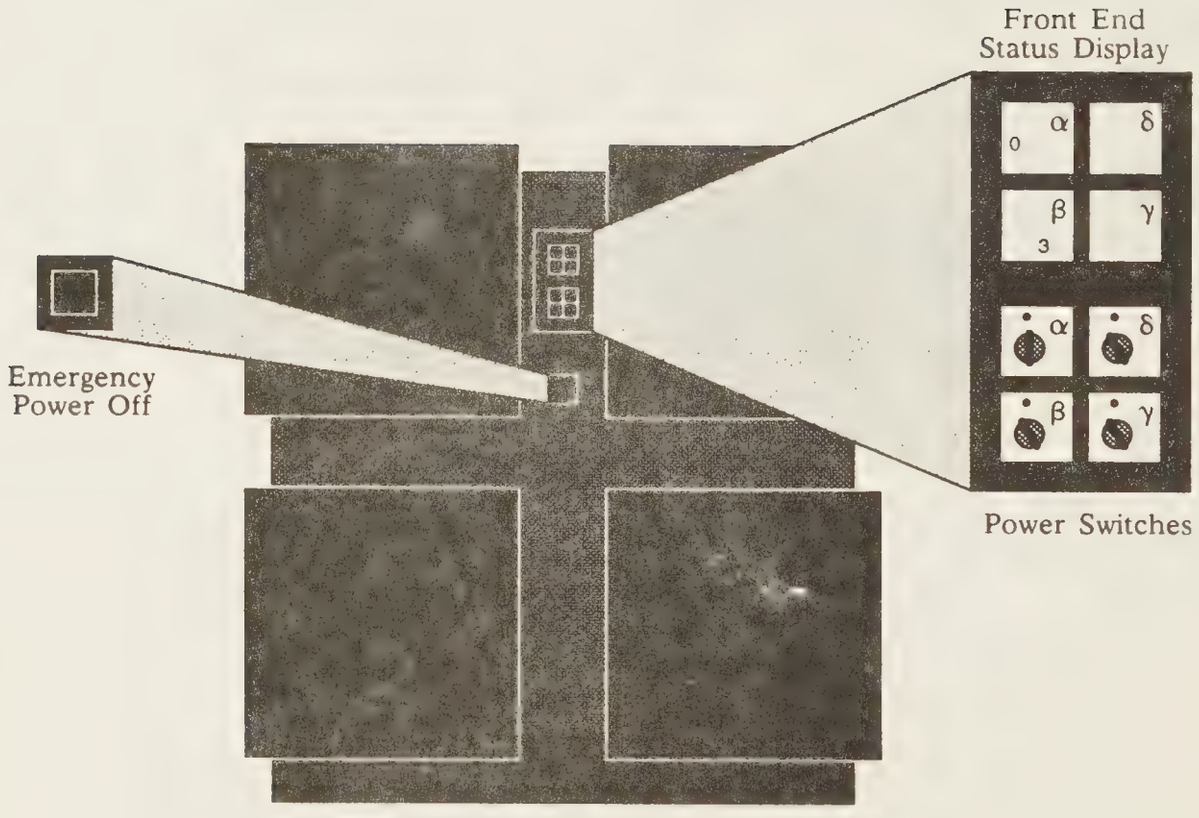


Figure 2. Front Panel for 16,384 Processor Connection Machine

Using The Control Panel On A 16K Machine

Only one power switch, α, is used on a 16K processor system. That switch controls the power to the front left 16K processors. When the switch is turned clockwise to the vertical position, power is turned on, and an indicator light above the switch turns on to indicate that the power is on.

On a 16K processor system, the machine can be used as either a 16K processor system or as two 8K processor systems. The Front End Status Displays α and β show which front end system is connected to which section of the machine, α showing which front end the upper left 8K is connected to and β showing which section of the machine the bottom left 8K is connected to. These connections are displayed as single digits between 0 and 3, corresponding to each of the four front-end systems that can be attached to the Connection Machine system.

The Control Panel in Figure 2 shows a 16K machine, with power turned on. The top 8K section of the machine, α, is connected to front end number 0, and the bottom 8K section, β, is connected to front end number 3.

This scheme extends easily. On a machine with 32K processors, power switch δ turns on the front right 16K processors, and Front End Status Displays δ and γ show which front-end system is connected to the top and bottom right quarters.

Figure 3. Front Panel for 65,536 Processor Machine

Using The Control Panel On A 64K Machine

On a 64K processor system there is a slightly different control panel with slightly different operating instructions. The major visual difference is that the Greek letters on both the power switches and the status display are not in the same place they are on the 16K control panel. The power switches each control one 16K segment of the machine: α the front left, β the front right, γ the back left, and δ the back right. The status displays refer to the same processor arrangement, and show which front end is connected to which 16K segment. On the full 64K machine, the Connection Machine processors are allocated in groups of 16K; it is not possible to allocate an 8K unit.

In figure 3, Panel A shows a full 64K Connection Machine system attached to one single front end, identified as "0". Panel B shows a 64K system with a 32K segment attached to front end 0, and two 16K segments, one attached to front end 3 and one attached to front end 2.

Connection Machine segments are allocated to front-end processors in power of two increments. On the 64K machine, a front-end processor can be connected to an 16K, 32K, or 64K unit of the machine. There are rules that govern which segments of the machine can be combined together to make these units—for example, not any two 16K units can be combined to make a 32K unit. The two 16K processor segments α and β can be combined to make a 32K machine as can segments γ and δ. All other combinations are not supported.

Diagnostics

Diagnostics must be run from within the LISP environment. There are two types of whole system diagnostics that run on the Connection Machine hardware: one quick test that does a quick verification of the hardware, and one complete test that thoroughly exercises the machine. These diagnostics should be run any time a hardware problem is suspected. Warning: running these diagnostics destroys the contents of the memory in the attached Connection Machine processors.

One pass of the complete test takes several hours to run (a 16K processor machine executes the diagnostics faster than a 64K machine). This diagnostic should be run whenever the Connection Machine hardware is going to be idle for a significant amount of time, and at a minimum should be run every evening. With these diagnostics, even intermittent problems can be quickly detected.

The following are arguments to cmattach which are useful for diagnostics in pinning down the location of hardware problems:

:ucc0, **:ucc1**, **:ucc2**, or **:ucc3**

Exactly the specified microcontroller port is to be attached, regardless of whether that port controls 8,192 or 16,384 physical processors.

:ucc0-1, **:ucc2-3**, or **:ucc0-3**

Exactly the specified microcontroller ports (0 and 1, 2 and 3, or all four) are to be attached, regardless of the number of physical processors involved.

To execute the quick diagnostics, log onto the UNIX system and type:

```
% /usr/local/lisp-paris
(cm:hardware-test-fast)
```

This does a quick (less than five minutes) system verification of the attached hardware to assure that the macrocode set, microcode set, and hardware are in order. This test can also be run on the LISP machine, and it takes only 30 seconds; just type **(cm:hardware-test-fast)**. Warning: this test destroys all user memory state in the attached Connection Machine processors. This is in part because there are two types of microcode, diagnostics and PARIS, one of which may be loaded into the microcontrollers at a time. The PARIS microcode is used for all regular user operations; running diagnostics loads the `diagnostics` microcode into the microcontrollers. When diagnostics finish, the PARIS microcode will be active again. This is what will appear on the screen if no errors are detected:

```
Data: 1 2
Identity:
Select:
WELO/WEHI
ALU:
Flags:
Cond:
NEWS: Bypassing NEWS for PINTB
Cube
T
```

To execute the thorough diagnostics, log onto the LISP machine and type:

(cm:hardware-test-complete)

This does a thorough system test of the attached hardware to assure that the macrocode set, microcode set, and hardware are in order. It takes between 3 and 3-1/2 hours on the LISP machine. Warning: this test destroys all user memory state in the attached Connection Machine processors. This operation can be called no matter what microcode is currently loaded; when it finishes, the PARIS microcode will be active. The function writes its output to both the screen and a file called `local:>cm>log>`*3600-name*`.log`. This form of the diagnostics runs continuously until aborted by the user.

Customer Support for System Administrators

The most effective way to report a software problem is to send mail to the Arpanet address `bug-connection-machine@think`. This mailing list is monitored by a number of our technical people, and you will get a reply to your message within 48 hours Monday to Friday, be it an answer to the question or a note simply saying when we will be able to get you an answer.

If you encounter an error that halts your work, including malfunctioning hardware and software that has a glaring unworkable error, you should call our headquarters at (617) 876-1111 and ask for Customer Support. We prefer to get electronic mail, since more descriptive information can be included with the message. When sending a software problem report, be sure to include enough of the code that caused the problem so that we can reproduce the error.

Connection Machine Parallel Instruction Set

(PARIS)

The C Implementation

Release 4.0

Thinking Machines Corporation
Cambridge, Massachusetts

First printing, March 1987

Thinking Machines Corporation
245 First Street
Cambridge, MA 02142-1214
(617) 876-1111

Contents

List of Figures

Chapter 1
Introduction

PARIS is a low-level instruction set for programming the Connection Machine computer system. It is the lowest level protocol by which the actions of Connection Machine processors are directed by the front-end computer. PARIS is sometimes referred to as a *macro-instruction set* for the Connection Machine system because it is comparable in power to the (macro) instruction sets of typical sequential processors such as the VAX. Also, this term distinguishes it from the *micro-instruction set*, or microcode, executed by the Connection Machine system microcontroller, and the *nano-instruction set* that is directly executed by the individual hardware Connection Machine processors.

PARIS is intended primarily as a base upon which to build higher-level languages for the Connection Machine system. It provides a large number of operations similar to the machine-level instruction set of an ordinary computer. PARIS supports primitive operations on signed and unsigned integers and floating-point numbers, as well as message-passing operations and facilities for transferring data between the Connection Machine processors and the front-end computer.

Two versions of the PARIS user interface are currently provided, one for the C programming language and one for the LISP programming language. These interfaces are functionally identical; they differ only in conforming to the syntax and data types of the different languages. This description of PARIS presents the C syntax and assumes the use of the C language.

The user interface consists of a set of functions, which are PARIS instructions implemented as a library of C functions, and *global configuration variables*, both of which are accessible to the user's C programs. The C programs are compiled and executed on a UNIX front end; the PARIS functions called from within them direct the actions of the Connection Machine system by sending macro-instructions to the Connection Machine microcontroller. The global configuration variables contain information about the Connection Machine system, such as the number of processors available. Many PARIS operations are implemented directly by the microcontroller in microcode; such operations are mapped one-to-one onto microcontroller directives. Other PARIS operations are more complicated, and require some cooperation between the microcontroller and the library routines that run in the front-end computer.

The next section describes what is required in every C/PARIS program in order to use the PARIS instruction set. Chapters 2 through 4 explain the Connection Machine's virtual processor architecture, data formats, and operation formats and addressing modes. The PARIS instruction set is detailed in Chapters 5 through 10. For help in getting started using the Connection Machine from a UNIX system front end, see *Connection Machine User's Guide: Using a UNIX System Front End* in Volume II of The Connection Machine Documentation Set.

Initialization

#include <cm/cm.h>

Every C program containing PARIS instructions must have the above line at the beginning of the program. This statement sets up the C/PARIS programming environment by including the file defining the data types used by PARIS functions, `CM_memaddr_t` and `CM_cubeaddr_t`, and the global configuration variables.

CM_init()

This initialization function must appear in every C/PARIS program before the first PARIS function. It initializes the values of the global configuration variables and *warm boots* the Connection Machine. This function has no arguments.

Warm booting the Connection Machine consists of clearing the error status indicators for the attached Connection Machine hardware, clearing the input and output first-in first-out (FIFO) queues in the front end's bus interface. While the user memory areas in the Connection Machine system are not disturbed, they are checked for errors and any memory errors are reported. Certain memory areas in the Connection Machine system are re-initialized, but the state of the pseudo-random number generator is not altered and the system lights display mode is not altered.

C/PARIS Program Template:

```
#include <cm/cm.h>
main()
{
   CM_init()
   [C code]
   [PARIS function calls occurring within C code]
   [C code]
}
```

Chapter 2
Virtual Machine Architecture

An important property of the Connection Machine architecture is *scalability*. At present a single Connection Machine system can have 16,384 or 32,768 or 65,536 physical (hardware) processors, of which any single user can use a portion containing 8,192 or 16,384 or 32,768 or 65,536 processors. (See Figure 2–1. for an illustration of 65,536 processors.) In most cases the same software can be executed unchanged on Connection Machine systems (or portions) with different numbers of physical processors; the number of processors affects only the size of the problem that can be handled.

PARIS enhances this scalability by presenting to the user an abstract version of the Connection Machine hardware. The most important feature is the *virtual processor* facility, whereby each physical processor is used to simulate some number of virtual processors. The point is that a program can be written assuming any appropriate number of processors; these virtual processors are then mapped onto physical processors. In this way a program can be executed unchanged on Connection Machine systems with different numbers of physical processors, even if it requires a certain minimum number of processors, with an essentially linear trade-off between number of physical processors and execution time. (There is a memory trade-off as well: the memory of a physical processor is divided equally among the virtual processors it supports.)

The Connection Machine hardware supports two mechanisms for interprocessor communication. The more general mechanism is the *router*, which allows data to be sent from any processor directly to any other processor; indeed, many processors can send data to many other processors simultaneously. The less general mechanism is redundant, but optimizes an important special case for speed. It organizes the processors as a two-dimensional grid and allows every processor to send data to an immediate neighbor in one of four directions, labeled North, East, West, and South; this mechanism is called the NEWS grid, from the initials of the four directions. Using these hardware mechanisms, PARIS provides identical virtual mechanisms within the virtual processor framework.

The Connection Machine hardware provides a very primitive instruction set, where each instruction operates on only a few bits per processor. PARIS implements a rich virtual instruction set, including arithmetic on integer and floating-point number representations. PARIS also supports a per-processor stack, indexed by a global (not per-processor) stack pointer. The PARIS instruction set is comparable in expressive power to the assembly language of an ordinary computer, except that a single PARIS instruction can call for parallel execution of many copies of an operation, one per processor.

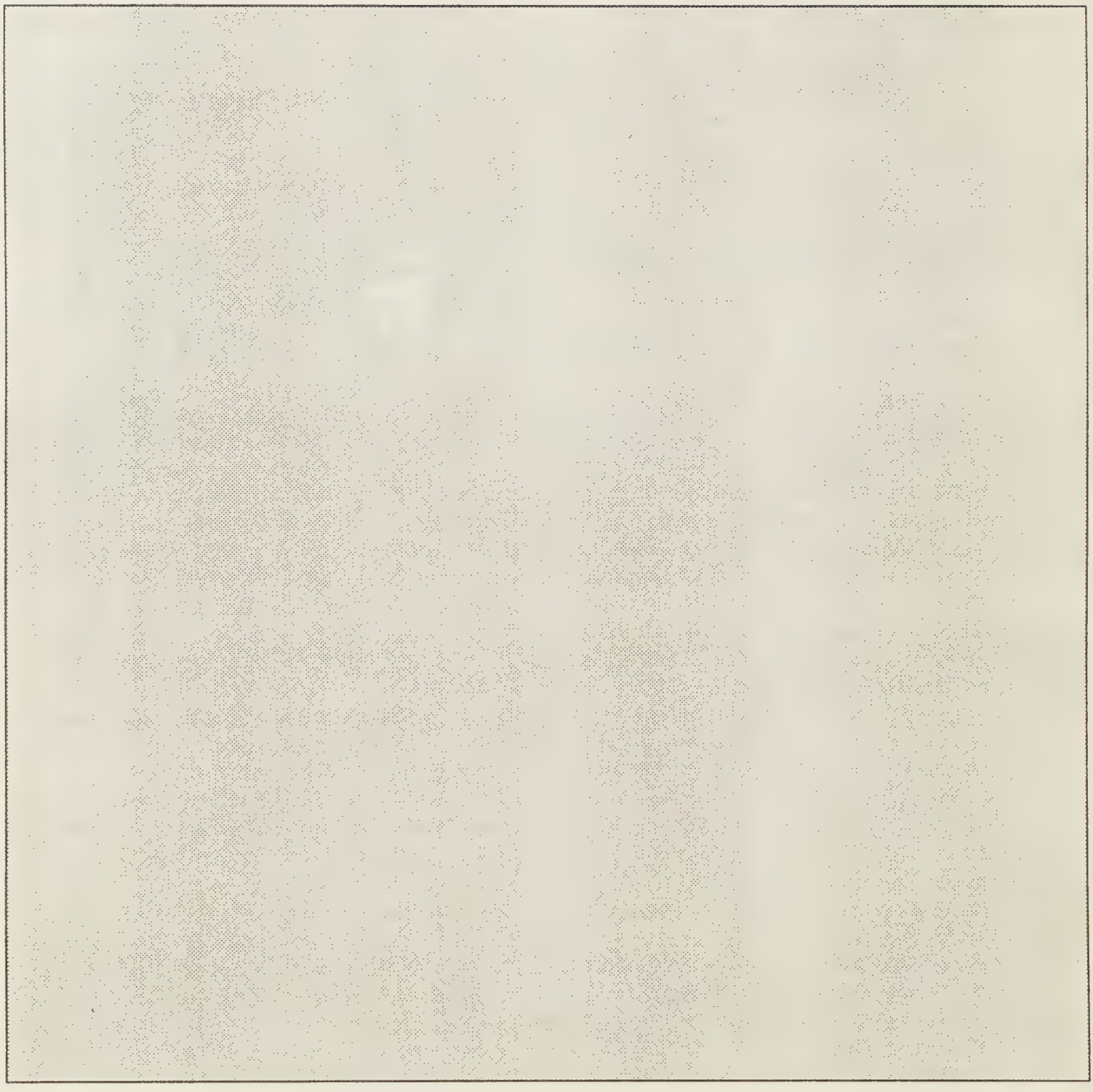


Figure 2–1. 65,536 processors

Virtual Processor Organization

When a PARIS program is to be run on the Connection Machine system, certain initialization operations must be performed first. Among other things, the virtual processor configuration must be defined. Two numbers specify the number of virtual processors per physical processor. Each physical processor simulates a small two-dimensional grid of virtual processors; the two numbers indicate the size of this grid in the x (east-west) and y (north-south) directions. These small grids are then implicitly joined together to make one large NEWS grid, in exactly the same way that the physical processors are arranged as a two-dimensional grid. The global variables `CM_x_virtual_to_physical_processor_ratio` and `CM_y_virtual_to_physical_processor_ratio` are set by the initialization program to the size of the small grid in the x and y directions.

For example, suppose that the numbers 2 and 4 are specified. Then each physical processor will simulate 8 virtual processors arranged in a 2×4 grid. The 65,536 physical processors of an entire Connection Machine system are arranged in a 128×512 grid, so the entire virtual grid will be of size

$$(2 \times 128) \times (4 \times 512) = 256 \times 2048$$

for a total of 524,288 (512K) virtual processors.

One might instead specify the numbers 8 and 2. Then each physical processor will simulate 16 virtual processors. The entire virtual grid will be of size

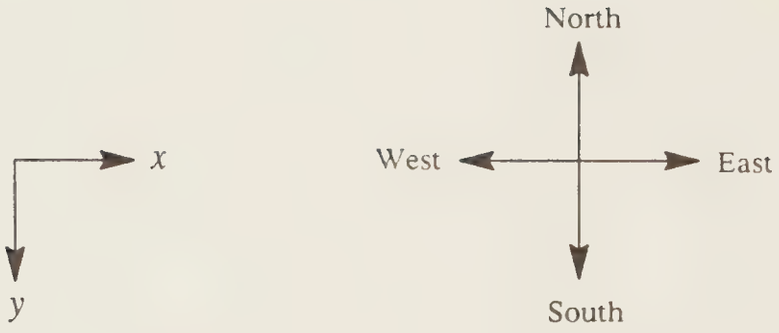
$$(8 \times 128) \times (2 \times 512) = 1024 \times 1024$$

for a total of 1,048,576 (1M) virtual processors.

If the numbers 1 and 1 are specified, then there is one virtual processor per physical processor; as far as execution speed is concerned, it is as if virtual processors were not in use.

Each virtual processor can be identified by its NEWS coordinates (x, y). The x coordinate increases toward the east, and the y coordinate increases toward the south, *not* toward the north. This convention is chosen to be compatible with the coordinate system used for raster-scan display terminals, where north is up, south is down, east is right, and west is left; x increases toward the right and y increases downward. See Figure 2–2.

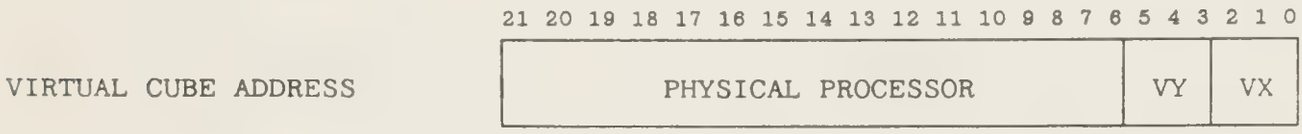
Virtual processors can communicate not only via the virtual NEWS grid, but also via message-sending through the Connection Machine routers. Every virtual processor (like every physical processor) is labeled by a distinct integer called its *cube address*, or sometimes just *address*. Virtual processor addresses are in general longer than physical processor addresses, but otherwise behave in much the same way. The global variable `CM_cube_address_length` specifies how many bits are needed to represent a virtual processor address. This number is equal to the number of bits per physical processor address plus $\log_2 x + \log_2 y$, where x and y are the values of `CM_x_virtual_to_physical_processor_ratio` and `CM_y_virtual_to_physical_processor_ratio`. These values must always be integral powers of two.



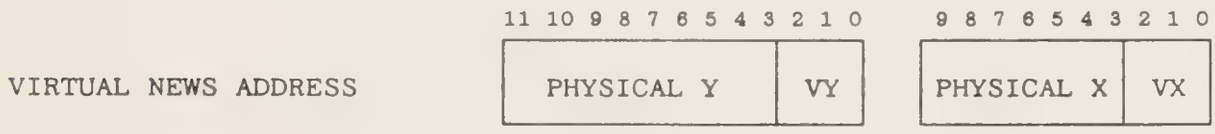
0,0	1,0	2,0	3,0	4,0	5,0	6,0	7,0	8,0	...	127,0
0,1	1,1	2,1	3,1	4,1	5,1	6,1	7,1	8,1	...	127,1
0,2	1,2	2,2	3,2	4,2	5,2	6,2	7,2	8,2	...	127,2
0,3	1,3	2,3	3,3	4,3	5,3	6,3	7,3	8,3	...	127,3
0,4	1,4	2,4	3,4	4,4	5,4	6,4	7,4	8,4	...	127,4
0,5	1,5	2,5	3,5	4,5	5,5	6,5	7,5	8,5	...	127,5
0,6	1,6	2,6	3,6	4,6	5,6	6,6	7,6	8,6	...	127,6
0,7	1,7	2,7	3,7	4,7	5,7	6,7	7,7	8,7	...	127,7
0,8	1,8	2,8	3,8	4,8	5,8	6,8	7,8	8,8	...	127,8
0,9	1,9	2,9	3,9	4,9	5,9	6,9	7,9	8,9	...	127,9
0,10	1,10	2,10	3,10	4,10	5,10	6,10	7,10	8,10	...	127,10
0,11	1,11	2,11	3,11	4,11	5,11	6,11	7,11	8,11	...	127,11
0,12	1,12	2,12	3,12	4,12	5,12	6,12	7,12	8,12	...	127,12
0,13	1,13	2,13	3,13	4,13	5,13	6,13	7,13	8,13	...	127,13
0,14	1,14	2,14	3,14	4,14	5,14	6,14	7,14	8,14	...	127,14
0,15	1,15	2,15	3,15	4,15	5,15	6,15	7,15	8,15	...	127,15
0,16	1,16	2,16	3,16	4,16	5,16	6,16	7,16	8,16	...	127,16
0,17	1,17	2,17	3,17	4,17	5,17	6,17	7,17	8,17	...	127,17
0,18	1,18	2,18	3,18	4,18	5,18	6,18	7,18	8,18	...	127,18
.	.	.	.	.	.	.	.	.		.
.	.	.	.	.	.	.	.	.		.
.	.	.	.	.	.	.	.	.		.
0,511	1,511	2,511	3,511	4,511	5,511	6,511	7,511	8,511	...	127,511

Figure 2–2. The NEWS coordinates for a 128 × 512 grid of processors. The *x* coordinate increases toward the east, and the *y* coordinate increases toward the south. This is the standard NEWS configuration when there are 65,536 physical processors and one virtual processor per physical processor.

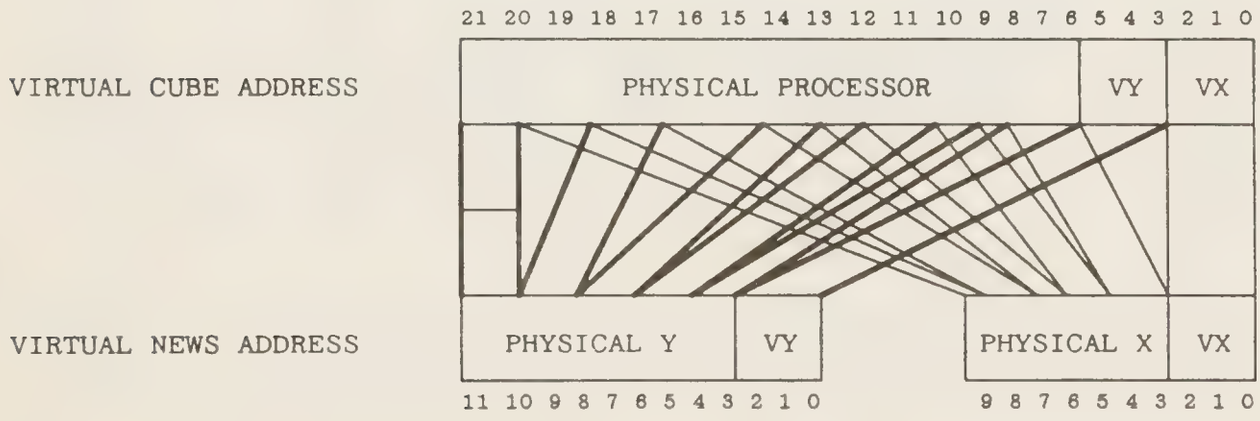
To be more specific, let us assume that a Connection Machine system with 65,536 physical processors is in use and that the virtual grid size is 8 × 8. Then 3 bits are needed to represent a virtual x address VX and 3 bits to represent a virtual y address VY within each processor. A virtual cube address is then laid out in this manner:



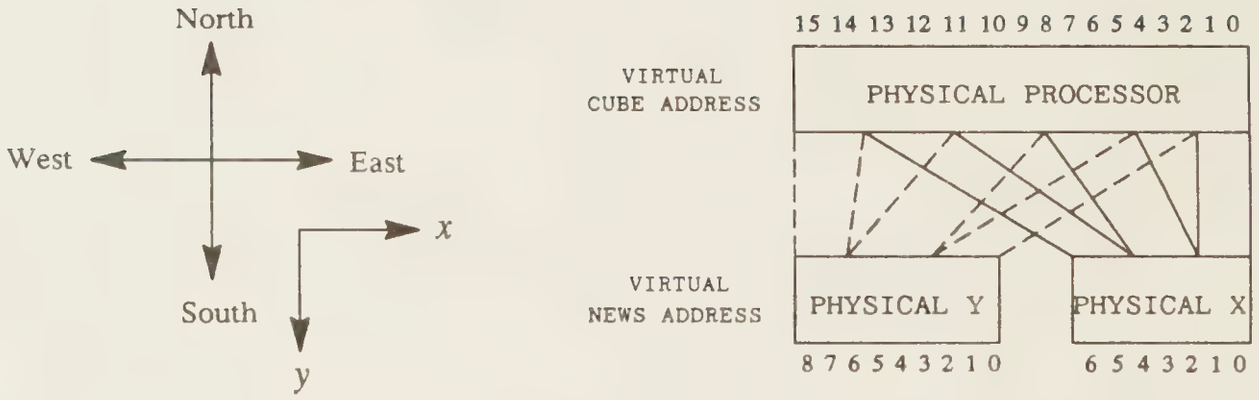
Furthermore virtual x and y addresses are laid out in this manner (the y address is shown on the left, and the x address on the right):



The mapping between the two looks like this:

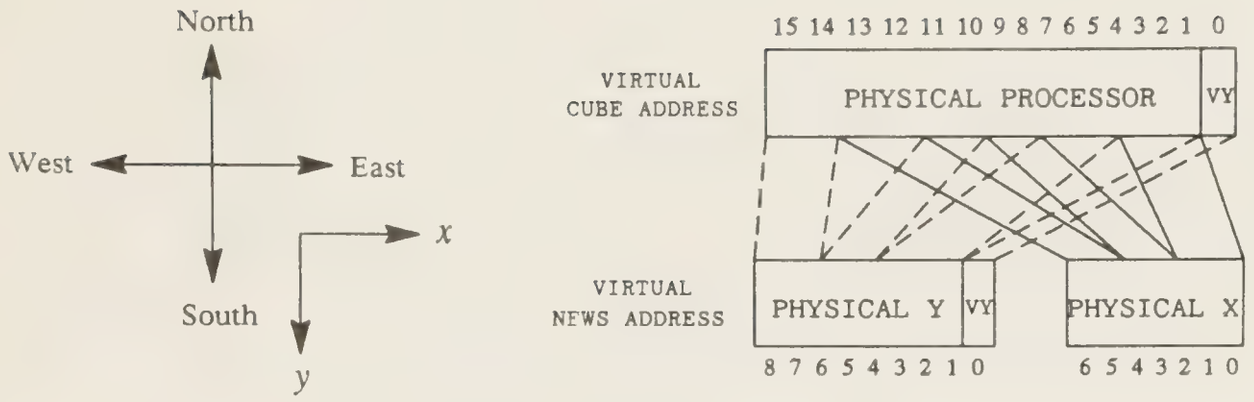


The permutation of virtual NEWS address bits that produces the virtual cube address is dependent on the number of physical processors being used to support the given number of virtual processors. (See Figures 2–3., 2–4., 2–5., and 2–6.) It is also dependent on the specific hardware implementations of the physical NEWS grid. It is therefore best to avoid dependence on the particular permutation; always use the operations `CM_x_from_cube`, `CM_y_from_cube`, and `CM_cube_from_x_y` to translate between cube addresses and NEWS addresses.



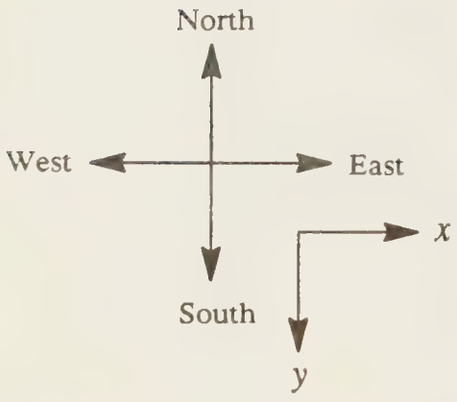
0	1	2	3	16	17	18	19	64	...	9811
4	5	6	7	20	21	22	23	68	...	9815
8	9	10	11	24	25	26	27	72	...	9819
12	13	14	15	28	29	30	31	76	...	9823
32	33	34	35	48	49	50	51	96	...	9843
36	37	38	39	52	53	54	55	100	...	9847
40	41	42	43	56	57	58	59	104	...	9851
44	45	46	47	60	61	62	63	108	...	9855
128	129	130	131	144	145	146	147	192	...	9939
132	133	134	135	148	149	150	151	196	...	9943
136	137	138	139	152	153	154	155	200	...	9947
140	141	142	143	156	157	158	159	204	...	9951
160	161	162	163	176	177	178	179	224	...	9971
164	165	166	167	180	181	182	183	228	...	9975
168	169	170	171	184	185	186	187	232	...	9979
172	173	174	175	188	189	190	191	236	...	9983
256	257	258	259	272	273	274	275	320	...	10067
260	261	262	263	276	277	278	279	324	...	10071
264	265	266	267	280	281	282	283	328	...	10075
.	.	.	.	.	.	.	.	.		.
.	.	.	.	.	.	.	.	.		.
.	.	.	.	.	.	.	.	.		.
55724	55725	55726	55727	55740	55741	55742	55743	55788	...	65535

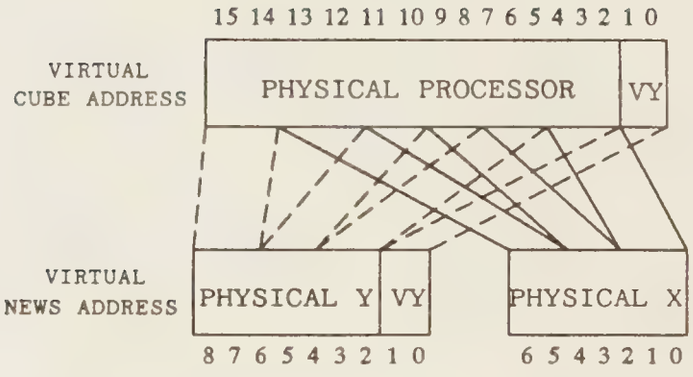
Figure 2–3. The cube addresses for a 128 × 512 grid of processors using 65,536 physical processors. Comparing this with Figure 2–2. reveals the mapping between cube addresses and NEWS coordinates in the case where 65,536 physical processors are used to contain a 128 × 512 NEWS grid. There is one virtual processor per physical processor. Each box enclosed in dashed lines represents one physical processor; thick lines delimit groups of 16 physical processors.



0	2	4	6	32	34	36	38	128	...	19622
1	3	5	7	33	35	37	39	129	...	19623
8	10	12	14	40	42	44	46	136	...	19630
9	11	13	15	41	43	45	47	137	...	19631
16	18	20	22	48	50	52	54	144	...	19638
17	19	21	23	49	51	53	55	145	...	19639
24	26	28	30	56	58	60	62	152	...	19646
25	27	29	31	57	59	61	63	153	...	19647
64	66	68	70	96	98	100	102	192	...	19686
65	67	69	71	97	99	101	103	193	...	19687
72	74	76	78	104	106	108	110	200	...	19694
73	75	77	79	105	107	109	111	201	...	19695
80	82	84	86	112	114	116	118	208	...	19702
81	83	85	87	113	115	117	119	209	...	19703
88	90	92	94	120	122	124	126	216	...	19710
89	91	93	95	121	123	125	127	217	...	19711
256	257	258	259	272	273	274	275	384	...	19878
260	261	262	263	276	277	278	279	385	...	19879
264	265	266	267	280	281	282	283	392	...	19886
.	.	.	.	.	.	.	.	.		.
.	.	.	.	.	.	.	.	.		.
.	.	.	.	.	.	.	.	.		.
45913	45915	45917	45919	45945	45947	45949	45951	46041	...	65535

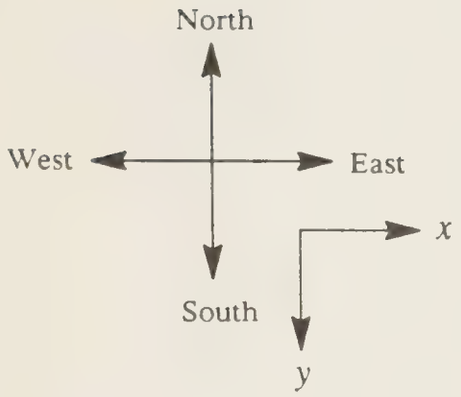
Figure 2-4. The cube addresses for a 128 × 512 grid of processors using 32,768 physical processors. In this case there is a 1 × 2 grid of virtual processors per physical processor. Each box enclosed in dashed lines represents one physical processor; thick lines delimit groups of 16 physical processors.

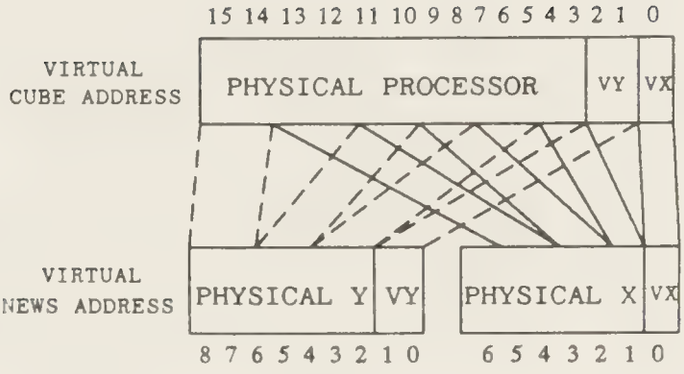




0	4	8	12	64	68	72	76	256	...	39244
1	5	9	13	65	69	73	77	257	...	39245
2	6	10	14	66	70	74	78	258	...	39246
3	7	11	15	67	71	75	79	259	...	39247
16	20	24	28	80	84	88	92	272	...	39260
17	21	25	29	81	85	89	93	273	...	39261
18	22	26	30	82	86	90	94	274	...	39262
19	23	27	31	83	87	91	95	275	...	39263
32	36	40	44	96	100	104	108	288	...	39276
33	37	41	45	97	101	105	109	289	...	39277
34	38	42	46	98	102	106	110	290	...	39278
35	39	43	47	99	103	107	111	291	...	39279
48	52	56	60	112	116	120	124	304	...	39292
49	53	57	61	113	117	121	125	305	...	39293
50	54	58	62	114	118	122	126	306	...	39294
51	55	59	63	115	119	123	127	307	...	39295
128	132	136	140	192	196	200	204	384	...	39372
129	133	137	141	193	197	201	205	385	...	39373
130	134	138	142	194	198	202	206	386	...	39374
.	.	.	.	.	.	.	.	.		.
.	.	.	.	.	.	.	.	.		.
.	.	.	.	.	.	.	.	.		.
26291	26295	26299	26303	26355	26359	26363	26367	26547	...	65535

Figure 2–5. The cube addresses for a 128 × 512 grid of processors using 16,384 physical processors. In this case there is a 1 × 4 grid of virtual processors per physical processor. Each box enclosed in dashed lines represents one physical processor; thick lines delimit groups of 16 physical processors.





0	1	8	9	16	17	24	25	128	...	12953
2	3	10	11	18	19	26	27	130	...	12955
4	5	12	13	20	21	28	29	132	...	12957
6	7	14	15	22	23	30	31	134	...	12959
32	33	40	41	48	49	56	57	160	...	12985
34	35	42	43	50	51	58	59	162	...	12987
36	37	44	45	52	53	60	61	164	...	12989
38	39	46	47	54	55	62	63	166	...	12991
64	65	72	73	80	81	88	89	192	...	13017
66	67	74	75	82	83	90	91	194	...	13019
68	69	76	77	84	85	92	93	196	...	13021
70	71	78	79	86	87	94	95	198	...	13023
96	97	104	105	112	113	120	121	224	...	13049
98	99	106	107	114	115	122	123	226	...	13051
100	101	108	109	116	117	124	125	228	...	13053
102	103	110	111	118	119	126	127	230	...	13055
256	257	264	265	272	273	280	281	384	...	13209
258	259	266	267	274	275	282	283	386	...	13211
260	261	268	269	276	277	284	285	388	...	13213
.	.	.	.	.	.	.	.	.		.
.	.	.	.	.	.	.	.	.		.
.	.	.	.	.	.	.	.	.		.
52582	52583	52590	52591	52698	52699	52806	52807	64710	...	65535

Figure 2–6. The cube addresses for a 128 × 512 grid of processors using 8,192 physical processors. In this case there is a 2 × 4 grid of virtual processors per physical processor. Each box enclosed in dashed lines represents one physical processor; thick lines delimit groups of 16 physical processors.

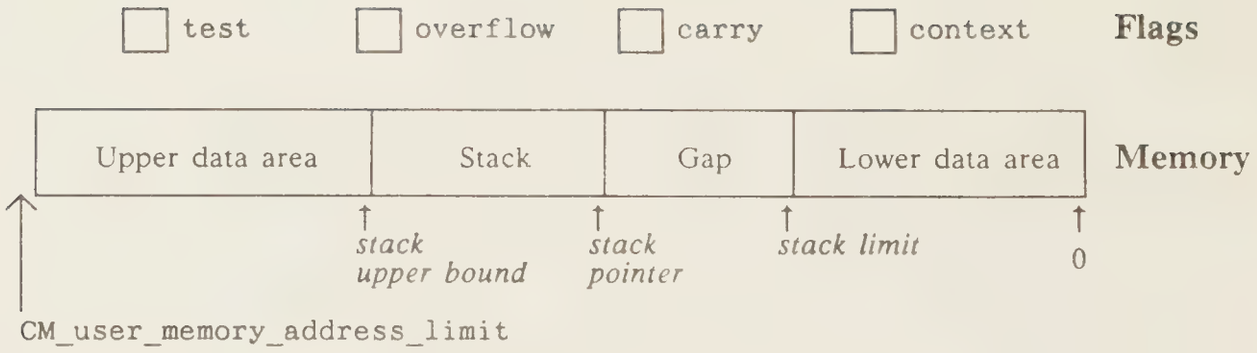


Figure 2–7. Virtual Processor Memory Layout and Flags

Memory

Each virtual processor has some amount of memory, addressable down to individual bits. Within each virtual processor the bit addresses run from 0 (inclusive) to some limit *m* (exclusive); the limit *m* is the value of the global variable `CM_user_memory_address_limit`. Typically the memory size *m* ranges from a few dozen to a few thousand bits, depending on how many virtual processors there are per physical processor.

In principle any part of this memory may be used for any purpose. However, PARIS conventionally divides the memory of each virtual processor into three regions: the *data area*, the *gap*, and the *stack*. The data area in turn may be divided into two parts, one below the stack and gap, and one above them. See Figure 2–7. The areas are delimited by the *stack limit*, the *stack pointer*, and the *stack upper bound*. The lower data area runs from address 0 (inclusive) to the stack limit (exclusive), and may be used for arbitrary purposes by the user. The gap runs from the stack limit (inclusive) to the stack pointer (exclusive), and is an expansion area; the stack implicitly grows into it as necessary. The data area may also be expanded into the gap, or reduced, returning space to the gap, by explicitly resetting the stack limit pointer using the `CM_set_stack_limit` operation. The stack runs from the stack pointer (inclusive) to the stack upper bound (exclusive); the upper bound is a boundary beyond which the stack should not be popped. The upper data area runs from the stack upper bound (inclusive) to the end of memory for that virtual processor. In many applications the upper data area is empty, but the existence of a stack upper bound distinct from the end of memory allows some flexibility in placing or moving around the stack area. In most applications the stack limit and stack upper bound are initialized once, or change only infrequently, and the stack pointer varies between them as items are pushed and popped. The stack pointer points to the "top" of the stack (which is at the lower-addressed end of the stack area).

Stack

The stack begins near the high end of memory within each virtual processor and grows downward. Most PARIS operations take operands from absolute memory locations. Stack-relative addressing may also be used, wherein an operand is addressed relative to the stack pointer (but no implicit pushing or popping is performed). There is a single global stack

pointer, rather than one per processor. A number of operations are provided for moving the stack pointer and changing the stack limit and stack upper bound; see Chapter 7.

A few PARIS operations require the implicit use of the stack (actually, the gap) for temporary scratch memory:

```
CM_array_ref                    CM_array_set
CM_global_add                   CM_max_scan
CM_add_scan                     CM_u_max_scan
CM_processor_cons               CM_f_max_scan
CM_store_with_overwrite         CM_get_from_north
CM_store_with_logior            CM_get_from_east
CM_store_with_logand            CM_get_from_west
CM_store_with_logxor            CM_get_from_south
CM_store_with_add               CM_rank
CM_store_with_max               CM_u_rank
CM_store_with_min               CM_f_rank
CM_store_with_u_max             CM_store
CM_store_with_u_min             CM_fetch
CM_get
```

Such operations will signal an error if the gap is not large enough to provide the required amount of temporary stack storage. See the individual descriptions of these operations for formulas describing the amount of storage required. Note that these operations may alter memory in the gap of any processor, whether selected or not.

Flags

Each PARIS virtual processor has an assortment of one-bit flags. Many PARIS operations store into these flags rather than, or in addition to, storing results into the memory. For example, the CM_add2 operation adds one signed integer to another, but also stores information into the carry flag and overflow flag. The PARIS programmer can refer to each flag by name. The entire set of flags for each virtual processor follows:

- CM_context_flag

  The context flag indicates which processors are currently active. Nearly all PARIS operations are *conditional*; the operation is effectively carried out only in those processors whose context flag is 1. Processors whose context flag is 0 are unaffected. The set of active processors—those whose context flag is 1—is commonly called "the currently selected set." Some operations are always unconditional. *All PARIS operations described below are conditional unless the individual description states otherwise.*

- CM_overflow_flag

  The overflow flag indicates which operations produced results that the destination field was too small to contain. Many PARIS operations can affect it. As a rule, integer operations set or clear the overflow flag, while floating-point operations are "sticky" in that they either set it or leave it unchanged.

- CM_carry_flag

  The carry flag holds the carry in and carry out for some arithmetic operations. The following operations can affect the carry flag:

CM_add2	CM_u_add2
CM_add_constant	CM_u_add_constant
CM_add_carry	CM_u_add_carry
CM_add_flags	CM_u_add_flags
CM_subtract2	CM_u_subtract2
CM_subtract_constant	CM_u_subtract_constant
CM_subtract_borrow	CM_u_subtract_borrow

  Only the operations CM_add_carry, CM_u_add_carry, CM_subtract_borrow, and CM_u_subtract_borrow use the carry flag as an implicit input.

- CM_test_flag

  The test flag holds the result of numeric comparisons and other tests, or indicates which operations failed because of bad operands. The following operations can affect the test flag:

		CM_f_divide
CM_isqrt		CM_f_sqrt
CM_isminus		CM_f_isminus
CM_isplus	CM_u_isplus	CM_f_isplus
CM_iszero	CM_u_iszero	CM_f_iszero
CM_eq	CM_u_eq	CM_f_eq
CM_ne	CM_u_ne	CM_f_ne
CM_lt	CM_u_lt	CM_f_lt
CM_le	CM_u_le	CM_f_le
CM_gt	CM_u_gt	CM_f_gt
CM_ge	CM_u_ge	CM_f_ge
CM_max	CM_u_max	CM_f_max
CM_min	CM_u_min	CM_f_min
CM_max_constant	CM_u_max_constant	
CM_min_constant	CM_u_min_constant	

CM_eq_constant	CM_u_eq_constant
CM_ne_constant	CM_u_ne_constant
CM_lt_constant	CM_u_lt_constant
CM_le_constant	CM_u_le_constant
CM_gt_constant	CM_u_gt_constant
CM_ge_constant	CM_u_ge_constant

CM_floor_divide	CM_u_floor_divide
CM_ceiling_divide	CM_u_ceiling_divide
CM_truncate_divide	CM_u_truncate_divide
CM_round_divide	CM_u_round_divide

CM_mod	CM_u_mod
CM_rem	CM_u_rem
CM_floor_and_mod	CM_u_floor_and_mod
CM_ceiling_and_remainder	CM_u_ceiling_and_remainder
CM_truncate_and_rem	CM_u_truncate_and_rem
CM_round_and_remainder	CM_u_round_and_remainder
CM_global_max	CM_global_u_max
CM_global_min	CM_global_u_min
CM_array_ref	CM_array_set
CM_send_with_overwrite	CM_store_with_overwrite
CM_send_with_logior	CM_store_with_logior
CM_send_with_logand	CM_store_with_logand
CM_send_with_logxor	CM_store_with_logxor
CM_send_with_add	CM_store_with_add
CM_send_with_max	CM_store_with_max
CM_send_with_min	CM_store_with_min
CM_send_with_u_max	CM_store_with_u_max
CM_send_with_u_min	CM_store_with_u_min
CM_send	CM_store

Chapter 3
Data Formats

The memory of a Connection Machine processor may be regarded as a simple linear sequence of bits, with no particular alignment or grouping characteristics. A data item always consists of a string of bits having consecutive addresses within the memory of a processor. Such a bit string is called a *field*.

Many PARIS operations may be regarded as interpreting bit fields as being of particular data types or formats. Currently PARIS supports three data types besides ordinary bit fields:

- signed integers, represented in two's-complement format
- unsigned integers, represented in straight binary format
- floating-point numbers, represented in a format close to that specified by IEEE standard 754 for floating-point arithmetic

The Connection Machine system allows unusual flexibility in that the hardware does not enforce any particular length or alignment requirements. PARIS supports integers and floating-point numbers of almost any size, although one limitation is that data exchanged with the front-end computer must comply with the front end's format requirements.

Most PARIS operations operate on fields within a processor, delivering results to other fields within that processor. Frequently we speak of one data item, but really mean to speak of many instances of that data item, one for each selected processor, to be considered or operated on in parallel. For example, when we say that an operation sets a flag when a field has such-and-so value, we mean that a separate decision is made within each processor whether to set that processor's flag, based on the value of the field within that processor.

Bit Fields

A bit field is specified by a bit address a and a positive length n; the field consists of the bits with addresses a through $a + n - 1$, inclusive. Therefore the address of a field is the same as that of the lowest-addressed bit.

Some PARIS operations can operate on the flags as if they were bit fields of length 1 residing in memory. The permitted operations on flags are described in Chapter 4.

Signed Integers

A signed integer is specified in the same way as a simple bit field, by a bit address a and a positive length n. The signed integer is represented in two's-complement form, and so a signed integer of length n can take on values in the range $-(2^{(n-1)})$ through $2^{n-1} - 1$, inclusive. The least significant bit has address a, and the most significant (sign) bit has address $a + n - 1$.

All arithmetic on signed integers is performed in a strict wraparound mode. As a rule, if the result of an operation overflows the destination field, the `overflow` flag is set, and the desti-

nation receives as many low-order bits of the true result as will fit. For example, using 4-bit signed arithmetic, multiplying 4 by −7 will produce the 4-bit result 4 (and also set the `overflow` flag), because the two's-complement representation of −28 is ...111111100100, of which the four low-order bits are 0100, or 4. Signed integer operations that do not overflow clear the `overflow` flag.

In order to simplify the Connection Machine microcode, this arbitrary restriction is imposed: the length n may not be zero or one. In addition, certain operations on signed integers cannot handle operands whose length is greater than the value of the variable `CM_maximum_integer_length`; see the section entitled "Implementation Restrictions."

Unsigned Integers

An unsigned integer is specified in the same way as a simple bit field, by a bit address a and a positive length n. The unsigned integer is represented in straight binary form, and so an unsigned integer of length n can take on values in the range 0 through $2^n - 1$, inclusive. The least significant bit has address a, and the most significant bit has address $a + n - 1$.

All arithmetic on unsigned integers is performed in a strict wraparound mode, modulo 2^n. As a rule, if the result of an operation overflows the destination field, the `overflow` flag is set, and the destination receives as many low-order bits of the true result as will fit. For example, using 4-bit unsigned arithmetic, multiplying 4 by 7 will produce the 4-bit result 12 (and also set the `overflow` flag), because the two's-complement representation of 28 is ...00000011100, of which the four low-order bits are 1100, or 12. Unsigned integer operations that do not overflow clear the `overflow` flag.

Unsigned integers, unlike signed integers, may be of length zero or one as well as of larger sizes. However, certain operations on unsigned integers cannot handle operands whose length is greater than the value of the variable `CM_maximum_integer_length`; see the section entitled "Implementation Restrictions."

Floating-Point Numbers

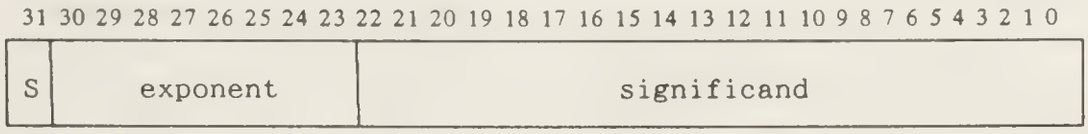
A floating-point data item is specified by three parameters: a bit address a, a significand length s, and an exponent length e. The total number of bits in the representation is $s + e + 1$, and the data item occupies the bits with addresses a through $a + s + e$, inclusive.

The significand occupies bits a through $a + s - 1$, with the least significant bit at address a. A hidden-bit representation is used, and so the significand is normally interpreted as having a 1-bit as its most significant bit implicitly just above the bit at address $a + s - 1$. If the exponent field is all zero-bits, however, then the hidden bit is taken to be 0.

The exponent occupies bits $a + s$ through $a + s + e - 1$, with the least significant bit at address $a + s$. An excess-$(2^{e-1} - 1)$ representation is used.

The sign bit occupies bit $a + s + e$, and is 1 for a negative number and 0 for a positive number. Overall, a sign-magnitude representation is used, so inverting the sign of a floating-point number merely involves flipping the sign bit. Note that there is both a plus zero and a minus zero.

When $s = 23$ and $e = 8$, this is equivalent to the IEEE standard single-precision format, which looks like this:



All of the PARIS floating-point operations default their floating-point length parameters to this format. When $s = 52$ and $e = 11$, the PARIS floating-point format is equivalent to IEEE standard double-precision format. The IEEE standard single-extended and double-extended formats can also be accommodated by suitable choices of s and e.

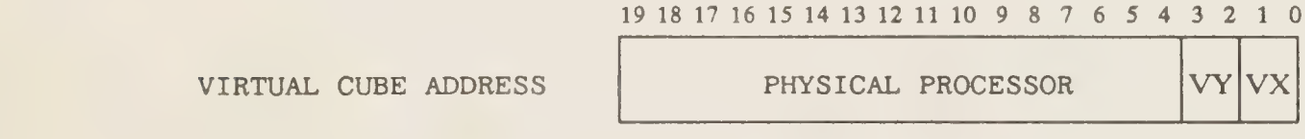
While the PARIS floating-point *format* is equivalent to the IEEE standard format, it must be emphasized that the PARIS implementation does not support equivalent *operations* at this time.[1] "Soft" underflow (using de-normalized numbers for the result) is not supported. Rounding is performed correctly in all cases, using the round-to-nearest mode; the several rounding modes are not supported. The not-a-number (NAN) values are not supported. The standard exceptions and flags are not all supported. It is strongly recommended that a user of PARIS always use the IEEE standard formats unless careful analysis of the application (such as a need for speed or additional exponent range) indicates that another format is required and adequate.

As a rule, if the result of an operation overflows, the `overflow` flag is set, and the destination exponent receives as many low-order bits of the true result exponent as will fit. Operations that do not overflow leave the `overflow` flag unaffected; they do not clear it.

In order to simplify the Connection Machine microcode, these arbitrary restrictions are imposed: $e \geq 2$, $s \geq 1$, and $2^{e-1} \geq s + 1$. In addition, certain operations on floating-point numbers cannot handle operands for which s exceeds the value of the variable `CM_maximum_significand_length` or e exceeds the value of the variable `CM_maximum_exponent_length`; see the section entitled "Implementation Restrictions."

Processor Addresses

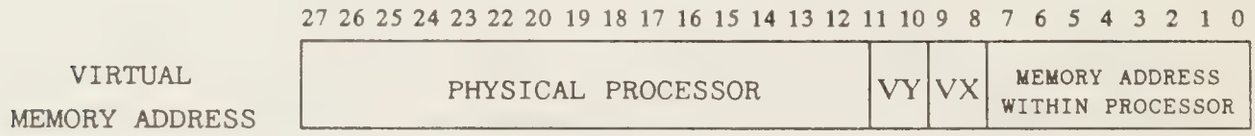
Two kinds of addresses are used in PARIS for general interprocessor communication. The first is the *virtual cube address*; such an address identifies an individual virtual processor. The number of bits in a virtual cube address is specified by the variable `CM_cube_address_length`. Such an address has this form for a 4 × 4 virtual NEWS configuration:



The second kind of address is the *virtual memory address*, which is the concatenation of a virtual cube address (in the more significant bits) and the address of a bit within the memory of the specified processor (in the less significant bits). A virtual memory address therefore

[1] Thinking Machines Corporation does intend to support all standard IEEE arithmetic operations in a future software release.

points to a specific bit or field within the entire Connection Machine system. Such a virtual memory address has this form for a 4 × 4 virtual NEWS configuration:



Such addresses usually reside in the memory of a Connection Machine processor as bit fields. To specify such an address to a PARIS operation, one gives the address (as a source or destination) of the bit field that contains the address.

Configuration Variables

The current configuration of the machine is reflected in global configuration variables. Programs refer to these so they can adapt to various sizes of machine. These variables are set whenever the Connection Machine is cold booted. They should never be set by the user, as there are many dependencies between them, which, if violated, will result in errors. Some variables are fixed by the hardware, while others depend on the arrangement of virtual processors set up by `cmcoldboot`.

Let p and q be the x and y dimensions, respectively, of the physical NEWS grid connecting all the physical processors. Let v and w be the x and y dimensions, respectively, of the overall virtual NEWS grid connecting all the virtual processors. Each of these four quantities is an integral power of two, and furthermore $v \geq p$ and $w \geq q$.

Let m be the amount of the memory in each virtual processor available to the user. (This amount will not necessarily be a power of two.) Every memory address a within a virtual processor must be in the range $0 \leq a < m$. The values of most of the configuration variables are expressible in terms of p, q, v, w, and m as shown below.

`CM_user_memory_address_limit`	$= m$
`CM_user_cube_address_limit`	$= v \cdot w$
`CM_full_virtual_memory_address_limit`	$= m \cdot v \cdot w$
`CM_user_x_dimension_limit`	$= v$
`CM_user_y_dimension_limit`	$= w$
`CM_physical_cube_address_limit`	$= p \cdot q$
`CM_physical_x_dimension_limit`	$= p$
`CM_physical_y_dimension_limit`	$= q$
`CM_virtual_to_physical_processor_ratio`	$= (v \cdot w)/(p \cdot q)$
`CM_x_virtual_to_physical_processor_ratio`	$= v/p$
`CM_y_virtual_to_physical_processor_ratio`	$= w/q$

These quantities are useful for measuring numbers of processors and memory locations.

CM_user_memory_address_length	$= \lceil \log_2 m \rceil$
CM_cube_address_length	$= \log_2 (v \cdot w)$
CM_virtual_memory_address_length	$= \lceil \log_2 m \cdot v \cdot w \rceil$
CM_x_news_address_length	$= \log_2 v$
CM_y_news_address_length	$= \log_2 w$
CM_physical_cube_address_length	$= \log_2 (p \cdot q)$
CM_physical_x_news_address_length	$= \log_2 p$
CM_physical_y_news_address_length	$= \log_2 q$
CM_purely_virtual_cube_address_length	$= \log_2 ((v \cdot w)/(p \cdot q))$
CM_purely_virtual_x_news_address_length	$= \log_2 (v/p)$
CM_purely_virtual_y_news_address_length	$= \log_2 (w/q)$

These quantities are the base two logarithms of the previously described quantities. They are useful for measuring the lengths (in bits) of fields containing processor numbers and memory addresses.

Implementation Restrictions

CM_maximum_integer_length

Because of implementation restrictions, a few operations on signed and unsigned integers cannot handle operands longer than the value of CM_maximum_integer_length. These operations are:

CM_isqrt	CM_u_isqrt
CM_float	CM_u_float
CM_multiply2	CM_u_multiply2
CM_multiply	CM_u_multiply
CM_floor_divide	CM_u_floor_divide
CM_ceiling_divide	CM_u_ceiling_divide
CM_truncate_divide	CM_u_truncate_divide
CM_round_divide	CM_u_round_divide
CM_mod	CM_u_mod
CM_rem	CM_u_rem
CM_floor_and_mod	CM_u_floor_and_mod
CM_ceiling_and_remainder	CM_u_ceiling_and_remainder
CM_truncate_and_rem	CM_u_truncate_and_rem
CM_round_and_remainder	CM_u_round_and_remainder
CM_get_from_north	CM_get_from_north_always
CM_get_from_east	CM_get_from_east_always
CM_get_from_west	CM_get_from_west_always
CM_get_from_south	CM_get_from_south_always

Experimentation might reveal that in certain cases some of these operations succeed when applied to operands that are longer than this variable, but that fact is not guaranteed in succeeding software releases.

The value of CM_maximum_integer_length is always at least as great as the smaller of 128 and half the value of CM_user_memory_address_limit.

CM_maximum_significand_length
CM_maximum_exponent_length

Because of implementation restrictions, a few operations on floating-point numbers cannot handle operands with significands or exponents longer than a certain size. These operations are CM_f_add, CM_f_subtract, CM_f_multiply, CM_f_divide, and CM_f_sqrt.

Experimentation might reveal that in certain cases some of these operations succeed when applied to operands that are longer than specified by these variables, but that fact is not guaranteed in succeeding software releases.

The value of CM_maximum_significand_length is always at least as great as the smaller of 96 and eight less than half the value of CM_user_memory_address_limit.

The value of CM_maximum_exponent_length is always at least as great as the smaller of 32 and one fourth of the value of CM_user_memory_address_limit.

CM_maximum_message_length

Because of implementation restrictions, a few operations that send messages from one processor to another cannot handle operands longer than a certain size. These operations are:

CM_send_with_overwrite		CM_store_with_overwrite
CM_send_with_logior		CM_store_with_logior
CM_send_with_logand		CM_store_with_logand
CM_send_with_logxor		CM_store_with_logxor
CM_send_with_add		CM_store_with_add
CM_send_with_max		CM_store_with_max
CM_send_with_min		CM_store_with_min
CM_send_with_u_max		CM_store_with_u_max
CM_send_with_u_min		CM_store_with_u_min
CM_send		CM_store
CM_get		CM_fetch
CM_rank	CM_u_rank	CM_f_rank

All of these except the rank operations require the length of the data to be transmitted to be no greater than the value of CM_maximum_message_length. The rank operations have more complicated constraints; see their description for details.

Experimentation might reveal that in certain cases some of these operations succeed when applied to operands that are longer than specified by this variable, but that fact is not guaranteed in succeeding software releases.

The value of CM_maximum_message_length is always at least as great as the smaller of 192 and half the value of CM_user_memory_address_limit.

Front End Format Restrictions

Although the Connection Machine does not have any particular length requirements, there may be some format restrictions imposed by the front-end computer on data passed to and from the Connection Machine. For example, with a UNIX front end, CM_f_write_to_processor can only accept the C data type of double. Also the read/write array functions, which transfer signed and unsigned integers between the Connection Machine and the UNIX front end, will only handle 1-, 8-, 16-, and 32-bit array elements; the float versions of these functions handle only C types single and double.

Chapter 4
Operation Formats and Addressing Modes

PARIS operations are executed at the direction of a program running in the front-end machine. For each operation there is a function or macro that, when called, causes the Connection Machine hardware to perform the operation.

Memory Addresses and the Stack

Most PARIS operations operate on one or more bit fields. A bit field may be a string of bits in the memory or may be one of the flags.

In the following syntax description for bit-field-address operands, *x* is any C expression.

`CM_test_flag`	The operand is the `test` flag.
`CM_carry_flag`	The operand is the `carry` flag.
`CM_overflow_flag`	The operand is the `overflow` flag.
`CM_context_flag`	The operand is the `context` flag.
	This will, in effect, always be 1 unless the operation is unconditional.
`CM_stack(`*x*`)`	The expression *x* must be of type `unsigned`.
	The operand address is the stack pointer plus the value of *x*.
x	The expression *x* must be of type `unsigned`.
	The operand address is the value of *x*.

Example: to add the 16-bit signed integer starting at location 48 to that starting at location 80, storing the result at location 80:

```
CM_add2(80, 48, 16)
```

Example: to add 7 into the 16-bit signed integer starting at location 48:

```
CM_add_constant(48, 7, 16)
```

Example: to add the 13-bit field at location 72 to the 13-bit field on top of the stack:

```
CM_add2(CM_stack(0), 72, 13)
```

Example: to push a copy of the 13-bit field at location 51 onto the stack:

```
CM_push_space(13)
CM_move(CM_stack(0), 51, 13)
```

Example: suppose there are three fields on the top of the stack whose (common) length is calculated at run time; their length is in the C variable `truck_len`. To add the one that is third from the top to the one that is second from the top, leaving the topmost undisturbed:

```
CM_add2(CM_stack(truck_len), CM_stack(2*truck_len), truck_len)
```

Example: to pop a 13-bit field from the stack and store it at location 91:

```
CM_move(91, CM_stack(0), 13)
CM_pop_and_discard(13)
```

Unconditional Operations

Most PARIS operations are conditional: they take place only in processors that have a 1 in the context flag. Sometimes it is necessary to perform operations unconditionally (that is, without respect to the context flag). Some operations have unconditional versions, generally named by appending _always to the name of the conditional function. For example, CM_lognor_always is the unconditional equivalent of CM_lognor. These operations are:

CM_lognot_always	CM_move_always
CM_logand_always	CM_logior_always
CM_logxor_always	CM_logeqv_always
CM_lognand_always	CM_lognor_always
CM_logandc1_always	CM_logandc2_always
CM_logorc1_always	CM_logorc2_always
CM_get_from_north_always	CM_get_from_east_always
CM_get_from_west_always	CM_get_from_south_always

A few PARIS operations have only unconditional forms and do not necessarily have names ending in _always.

Example: to unconditionally set the context flag to 1 in every processor:

```
CM_move_constant_always(CM_context_flag, 1, 1)
```

(Note that the conditional version

```
CM_move_constant(CM_context_flag, 1, 1)
```

has no net effect because it is executed only in those processors whose context flag is already 1.)

Example: to unconditionally copy the overflow flag into the context flag:

```
CM_move_always(CM_context_flag,  CM_overflow_flag, 1)
```

Example: to unconditionally EXCLUSIVE OR the overflow flag into the context flag:

```
CM_logxor_always(CM_context_flag, CM_overflow_flag, 1)
```

Example: to unconditionally push the context flag onto the stack:

```
CM_push_space(1)
CM_move_always(CM_stack(0), CM_context_flag, 1)
```

Constant Operands

Certain operations accept as an operand a single datum computed within the front end that is broadcast to all of the Connection Machine processors as part of the operation. These are:

```
CM_add_constant                 CM_u_add_constant
CM_subtract_constant            CM_u_subtract_constant
CM_eq_constant                  CM_u_eq_constant
CM_ne_constant                  CM_u_ne_constant
CM_lt_constant                  CM_u_lt_constant
CM_le_constant                  CM_u_le_constant
CM_gt_constant                  CM_u_gt_constant
CM_ge_constant                  CM_u_ge_constant
CM_max_constant                 CM_u_max_constant
CM_min_constant                 CM_u_min_constant
CM_move_constant                CM_move_constant_always
CM_f_move_constant              CM_f_move_decoded_constant
```

For example, the second argument to the `CM_add_constant` operation is a constant operand; `CM_add_constant(43, 1, 32)` will, in those processors that are selected, add the value 1 to the 32-bit field starting at location 43. Similarly, `CM_lt_constant(43, 10000, 16)` will set the test flag in every selected processor in which the 16-bit field starting at location 43 contains a (signed) value that is less than ten thousand.

Flags as Operands

Certain operations allow the flags as operands. These are:

```
CM_move                         CM_move_always
CM_move_constant                CM_move_constant_always
CM_lognot                       CM_lognot_always
CM_logand                       CM_logand_always
CM_logior                       CM_logior_always
CM_logxor                       CM_logxor_always
CM_logeqv                       CM_logeqv_always
CM_lognand                      CM_lognand_always
CM_logandc1                     CM_logandc1_always
CM_logandc2                     CM_logandc2_always
CM_lognor                       CM_lognor_always
CM_logorc1                      CM_logorc1_always
CM_logorc2                      CM_logorc2_always
CM_get_from_north               CM_get_from_north_always
CM_get_from_east                CM_get_from_east_always
CM_get_from_west                CM_get_from_west_always
CM_get_from_south               CM_get_from_south_always
CM_global_count                 CM_global_count_always
CM_global_logand                CM_global_logand_always
CM_global_logior                CM_global_logior_always
CM_latch_leds                   CM_latch_leds_always
CM_u_read_from_processor        CM_u_write_to_processor
```

No other PARIS operation may legitimately be used to operate on the flags.

Example: to conditionally copy the overflow flag into the context flag:

```
CM_move(CM_context_flag, CM_overflow_flag, 1)
```

Example: to count all the selected processors that have the test flag set:

```
count == CM_global_count(CM_test_flag))
```

The CM_global_count operation returns the count as an integer.

Chapter 5
Simple Unary Operations

Each function in this chapter operates on one field and produces some result. The operation is performed in every processor selected by the context flag (CM_context_flag). In most cases some mathematical function of a *source* field is computed and stored in a *destination* field of the same length. The two fields are specified by three arguments: the *destination* address, the *source* address, and the common *length* of the two fields. (Sometimes the name *destination* is abbreviated to *dest*.)

For some operations, such as sign tests, the only effect is to set flags and there is no destination field as such.

For every function in this section, the same field may be used for both operand fields, and it will behave in the expected manner. Two fields are the same only if they are specified by identical argument values and also are treated by the function as having the same data type. The results are unpredictable if the two operand fields overlap but are not identical.

If a flag is not explicitly mentioned in the individual description of a function, then it leaves the flag unaffected.

Unary Operations on Bit Fields

CM_lognot(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

CM_lognot_always(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

Each bit of the *destination* field is set to the logical NOT of the corresponding bit of the *source* field.

CM_lognot_always is an unconditional version of CM_lognot.

Unary Operations on Signed Integers

CM_abs(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

The absolute value of the *source* is placed in the *destination*.

The overflow flag is set if the result cannot be represented in the destination field, and is cleared otherwise. Overflow occurs only when the source value is the most negative representable number.

CM_negate(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

The negative of the *source* is placed in the *destination*.

The overflow flag is set if the result cannot be represented in the destination field, and is cleared otherwise. Overflow occurs only when the source value is the most negative representable number.

CM_signum(destination, source, dlen, slen)
```
CM_memaddr_t destination, source;
unsigned dlen, slen;
```

The signed integer value −1, 0, or 1 is placed in the *destination* according to whether the *source* value is negative, zero, or positive, respectively. The destination field need not be the same length as the source field; 2 is a popular length.

CM_isqrt(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

The integer square root of the *source* value is placed in the *destination*; this is the largest integer not greater than the mathematical square root.

If the source value is negative, then the test flag is set and the destination value is unpredictable; otherwise the test flag is cleared.

The *length* must be not greater than the value of `CM_maximum_integer_length`.

CM_iszero(source, length)
```
CM_memaddr_t source;
unsigned length;
```

The test flag is set if the *source* field is zero, and is cleared otherwise.

CM_isminus(source, length)
```
CM_memaddr_t source;
unsigned length;
```

The test flag is set if the *source* field is negative, and is cleared otherwise.

CM_isplus(source, length)
```
CM_memaddr_t source;
unsigned length;
```

The test flag is set if the *source* field is positive, and is cleared otherwise.

CM_float(dest, source, slen, signif_len, expt_len)
```
CM_memaddr_t dest, source;
unsigned slen, signif_len, expt_len;
```

The *source* field, treated as a signed integer, is converted to a floating-point number and placed into *dest*. The length of the source field is specified by *slen*; the format of the destination field is specified by *signif_len* and *expt_len*.

The overflow flag is set if the source value cannot be represented in the destination field, and is unaffected otherwise.

The length *slen* must be not greater than the value of CM_maximum_integer_length.

Note that unlike almost all other PARIS calls, the source length *slen* precedes the destination lengths *signif_len* and *expt_len*. This matches the LISP implementation of PARIS.

CM_new_size(destination, source, dlen, slen)
```
CM_memaddr_t destination, source;
unsigned dlen, slen;
```

The *source* field, treated as a signed integer, is copied into the *destination* field.

If the length of the destination is equal to the length of the source, then the overflow flag is cleared.

If the length of the destination is greater than the length of the source, then the source field is sign-extended, and the overflow flag is cleared.

If the length of the destination is less than the length of the source, then overflow checking is performed. The overflow flag is set if the source value cannot be represented in the destination field, and is cleared otherwise.

CM_integer_length(destination, source, dlen, slen)
```
CM_memaddr_t destination, source;
unsigned dlen, slen;
```

The *destination* receives, as an *unsigned* integer, the result of the computation

$$\lceil(\log_2(s + 1))\rceil \qquad \text{if } s \geq 0$$

$$\lceil(\log_2(-s))\rceil \qquad \text{if } s < 0$$

where s is the *source* value. This quantity is one less than the minimum number of bits required to represent s as a signed number.

The overflow flag is set if the result cannot be represented in the destination field, and is cleared otherwise.

CM_logcount(destination, source, dlen, slen)
```
CM_memaddr_t destination, source;
unsigned dlen, slen;
```

The *destination* receives, as an *unsigned* integer, a count of the number of bits in the representation of the *source* value that are different from the sign bit.

The overflow flag is set if the result cannot be represented in the destination field, and is cleared otherwise.

Unary Operations on Unsigned Integers

CM_u_negate(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

The "unsigned negative" of the *source* (that is, the unsigned result of subtracting the source from zero) is placed in the *destination*.

The overflow flag is set if the source value is non-zero, and is cleared otherwise.

CM_u_isqrt(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

The integer square root of the unsigned *source* value is placed in the *destination*; this is the largest integer not greater than the mathematical square root.

The *length* must be not greater than the value of CM_maximum_integer_length.

CM_u_iszero(source, length)
```
CM_memaddr_t source;
unsigned length;
```

The test flag is set if the *source* field is zero, and is cleared otherwise. Of course, CM_u_iszero behaves identically to CM_iszero, and is provided primarily for reasons of symmetry.

CM_u_isplus(source, length)
```
CM_memaddr_t source;
unsigned length;
```

The test flag is set if the *source* field is non-zero, and is cleared otherwise.

CM_u_float(dest, source, slen, signif_len, expt_len)
```
CM_memaddr_t dest, source;
unsigned slen, signif_len, expt_len;
```

The *source* field, treated as an unsigned integer, is converted to a floating-point number and placed into *dest*. The length of the source field is specified by *slen*; the format of the destination field is specified by *signif_len* and *expt_len*.

The overflow flag is set if the source value cannot be represented in the destination field, and is unaffected otherwise.

The length *slen* must be not greater than the value of CM_maximum_integer_length.

Note that unlike almost all other calls, the source length *slen* precedes the destination lengths *signif_len* and *expt_len*. This matches the LISP implementation of PARIS.

CM_u_new_size(destination, source, dlen, slen)
```
CM_memaddr_t destination, source;
unsigned dlen, slen;
```

The *source* field, treated as an unsigned integer, is copied into the *destination* field.

If the length of the destination is equal to the length of the source, then the overflow flag is cleared.

If the length of the destination is greater than the length of the source, then the source field is zero-extended, and the overflow flag is cleared.

If the length of the destination is less than the length of the source, then overflow checking is performed. The overflow flag is set if the source value cannot be represented in the destination field, and is cleared otherwise.

CM_u_integer_length(destination, source, dlen, slen)
```
CM_memaddr_t destination, source;
unsigned dlen, slen;
```

The *destination* receives, as an unsigned integer, the value $\lceil \log_2(s + 1) \rceil$ where s is the *source* value. This quantity is the minimum number of bits required to represent s as an unsigned number.

The overflow flag is set if the source value cannot be represented in the destination field, and is cleared otherwise.

CM_u_logcount(destination, source, dlen, slen)
```
CM_memaddr_t destination, source;
unsigned dlen, slen;
```

The *destination* receives, as an unsigned integer, the number of bits in the representation of the *source* value that contain 1.

The overflow flag is set if the source value cannot be represented in the destination field, and is cleared otherwise.

CM_gray_code_from_integer(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

The *source* value is treated as an unsigned integer and converted to a Gray code representation, which is stored in the *destination*.

```
unsigned
```
CM_front_end_gray_code_from_integer(value)
```
unsigned value;
```

This operation is performed entirely within the front-end computer. The argument should be a non-negative integer; it is converted to a Gray code representation, which is returned.

CM_integer_from_gray_code(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

The *source* value is treated as a Gray code value and converted to an unsigned integer, which is stored in the *destination*.

```
unsigned
```
CM_front_end_integer_from_gray_code(gray_code)
```
unsigned gray_code;
```

This operation is performed entirely within the front-end computer. The argument should be a non-negative integer; it is treated as a Gray code value and converted to a non-negative integer, which is returned.

Unary Operations on Floating-Point Numbers

CM_f_abs(dest, source, signif_len, expt_len)
```
CM_memaddr_t dest, source;
unsigned signif_len, expt_len;
```

The absolute value of the *source* is placed into *dest*.

CM_f_negate(dest, source, signif_len, expt_len)
```
CM_memaddr_t dest, source;
unsigned signif_len, expt_len;
```

The negative of the *source* is placed into *dest*.

CM_f_signum(dest, source, dlen, signif_len, expt_len)
```
CM_memaddr_t dest, source;
unsigned dlen, signif_len, expt_len;
```

The signed integer value −1, 0, or 1 is placed into *dest* according to whether the *source* value is negative, zero, or positive, respectively. The length of the destination field is specified by *dlen*; the format of the source field is specified by *signif_len* and *expt_len*. The length of the destination field must be at least 2.

CM_f_float_signum(dest, source, signif_len, expt_len)
```
CM_memaddr_t dest, source;
unsigned signif_len, expt_len;
```

The floating-point value −1.0, −0.0, +0.0, or 1.0 is placed into *dest* according to whether the *source* value is negative, minus zero, plus zero, or positive, respectively.

CM_f_sqrt(dest, source, signif_len, expt_len)
```
CM_memaddr_t dest, source;
unsigned signif_len, expt_len;
```

The square root of the *source* value is placed into *dest*.

Note that the length *signif_len* must be not greater than the value of CM_maximum_significand_length, and the length *expt_len* must be not greater than the value of CM_maximum_exponent_length.

If the source value is negative (other than minus zero), then the test flag is set and the square root of the absolute value of the source is calculated. If the source value is minus zero, then the test flag is set and the result is minus zero. In all other cases the test flag is cleared.

CM_f_iszero(source, signif_len, expt_len)
```
CM_memaddr_t source;
unsigned signif_len, expt_len;
```

The test flag is set if the *source* field is zero (plus or minus), and is cleared otherwise.

CM_f_isminus(source, signif_len, expt_len)
```
CM_memaddr_t source;
unsigned signif_len, expt_len;
```

The test flag is set if the *source* field is negative (but not minus zero), and is cleared otherwise.

CM_f_isplus(source, signif_len, expt_len)
```
CM_memaddr_t source;
unsigned signif_len, expt_len;
```

The test flag is set if the *source* field is positive (but not plus zero), and is cleared otherwise.

CM_floor(dest, source, dlen, signif_len, expt_len)
```
CM_memaddr_t dest, source;
unsigned dlen, signif_len, expt_len;
```

CM_ceiling(dest, source, dlen, signif_len, expt_len)
```
CM_memaddr_t dest, source;
unsigned dlen, signif_len, expt_len;
```

CM_truncate(dest, source, dlen, signif_len, expt_len)
```
CM_memaddr_t dest, source;
unsigned dlen, signif_len, expt_len;
```

CM_round(dest, source, dlen, signif_len, expt_len)
```
CM_memaddr_t dest, source;
unsigned dlen, signif_len, expt_len;
```

The *source* field, treated as a floating-point number, is converted to a signed integer and placed into *dest*. The length of the destination field is specified by *dlen*; the format of the source field is specified by *signif_len* and *expt_len*.

The four operations differ only in the method of rounding when the source value is not an exact integer:

floor rounds toward –infinity.
ceiling rounds toward +infinity.
truncate rounds toward 0.
round rounds to the nearest integer, and if the source value lies exactly halfway between two integers, then the even integer is used.

The overflow flag is set if the source value cannot be represented in the destination field, and is cleared otherwise.

CM_u_floor(dest, source, dlen, signif_len, expt_len)
```
CM_memaddr_t dest, source;
unsigned dlen, signif_len, expt_len;
```

CM_u_ceiling(dest, source, dlen, signif_len, expt_len)
```
CM_memaddr_t dest, source;
unsigned dlen, signif_len, expt_len;
```

CM_u_truncate(dest, source, dlen, signif_len, expt_len)
```
CM_memaddr_t dest, source;
unsigned dlen, signif_len, expt_len;
```

CM_u_round(dest, source, dlen, signif_len, expt_len)
```
CM_memaddr_t dest, source;
unsigned dlen, signif_len, expt_len;
```

These are identical to the signed versions described above, but produce unsigned integer results.

The overflow flag is set if the *source* value cannot be represented in the *dest* field, and is cleared otherwise. Minus zero produces a zero result and does not set the overflow flag.

CM_f_new_size(dest, source, d_signif_len, d_expt_len, s_signif_len, s_expt_len)
```
CM_memaddr_t dest, source;
unsigned d_signif_len, d_expt_len, s_signif_len, s_expt_len;
```

The *source* field, treated as a floating-point number, is copied into the *dest* field in a new format. The format of the destination field is specified by *d_signif_len* and *d_expt_len*; the format of the source field is specified by *s_signif_len* and *s_expt_len*.

If the significand field of the destination is less than that of the source, then rounding is performed. This may lead to overflow.

If the exponent field of the destination is less than that of the source, then overflow may occur.

The overflow flag is set if the source value cannot be represented in the destination field, and is unaffected otherwise.

Chapter 6
Simple Binary Operations

Each function in this chapter operates on two fields. The operation is performed in every processor selected by the context flag. In most cases both operand fields are the same length, and a single *length* argument is provided.

Typically one operand field receives the result and also serves as the first input; it is called the *destination* even though it is also one source. The other operand field is called the *source*. The two fields are specified by three arguments *destination*, *source*, and the common *length* of the two fields. (Sometimes the name *destination* is abbreviated to *dest*.)

For some operations, such as relational comparisons, the only effect is to set flags and there is no destination field as such. In that case the two fields are specified by arguments called *source1*, *source2*, and *length*.

For every function in this section, the same field may be used for both *destination* and *source* arguments, and it will behave in the expected manner. The results are unpredictable if the two operand fields overlap but are not identical.

If a flag is not explicitly mentioned in the individual description of an function, then it leaves the flag unaffected.

Binary Operations on Bit Fields

All ten non-trivial Boolean operations on two operands are provided, in both conditional and unconditional versions.

CM_logand(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

CM_logand_always(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

Each bit of the *destination* field is set to the logical AND of that bit and the corresponding bit of the *source* field. The `CM_logand` function is conditional, and `CM_logand_always` is unconditional.

These functions may be used to operate on the flags.

CM_logior(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

CM_logior_always(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

Each bit of the *destination* field is set to the logical INCLUSIVE OR of that bit and the corresponding bit of the *source* field. The `CM_logior` function is conditional, and `CM_logior_always` is unconditional.

These functions may be used to operate on the flags.

CM_logxor(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

CM_logxor_always(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

Each bit of the *destination* field is set to the logical EXCLUSIVE OR of that bit and the corresponding bit of the *source* field. The CM_logxor function is conditional, and CM_logxor_always is unconditional.

These functions may be used to operate on the flags.

CM_logeqv(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

CM_logeqv_always(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

Each bit of the *destination* field is set to the logical EQUIVALENCE (also called EXCLUSIVE NOR) of that bit and the corresponding bit of the *source* field. The CM_logeqv function is conditional, and CM_logeqv_always is unconditional.

These functions may be used to operate on the flags.

CM_lognand(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

CM_lognand_always(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

Each bit of the *destination* field is set to the logical NAND (that is, the NOT of the AND) of that bit and the corresponding bit of the *source* field. The CM_lognand function is conditional, and CM_lognand_always is unconditional.

These functions may be used to operate on the flags.

CM_lognor(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

CM_lognor_always(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

Each bit of the *destination* field is set to the logical OR (that is, the NOT of the OR) of that bit and the corresponding bit of the *source* field. The CM_lognor function is conditional, and CM_lognor_always is unconditional.

These functions may be used to operate on the flags.

CM_logandc1(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

CM_logandc1_always(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

Each bit of the *destination* field is set to the logical AND of (1) the NOT of that bit, and (2) the corresponding bit of the *source* field. The `CM_logandc1` function is conditional, and `CM_logandc1_always` is unconditional.

These functions may be used to operate on the flags.

CM_logandc2(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

CM_logandc2_always(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

Each bit of the *destination* field is set to the logical AND of (1) that bit, and (2) the NOT of the corresponding bit of the *source* field. The `CM_logandc2` function is conditional, and `CM_logandc2_always` is unconditional.

These functions may be used to operate on the flags.

CM_logorc1(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

CM_logorc1_always(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

Each bit of the *destination* field is set to the logical OR of (1) the NOT of that bit, and (2) the corresponding bit of the *source* field. The `CM_logorc1` function is conditional, and `CM_logorc1_always` is unconditional.

These functions may be used to operate on the flags.

CM_logorc2(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

CM_logorc2_always(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

Each bit of the *destination* field is set to the logical OR of (1) that bit, and (2) the NOT of the corresponding bit of the *source* field. The `CM_logorc2` function is conditional, and `CM_logorc2_always` is unconditional.

These functions may be used to operate on the flags.

Binary Operations on Signed Integers

CM_add2(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

CM_add_constant(destination, source_constant, length)
```
CM_memaddr_t destination;
long source_constant;
unsigned length;
```

For CM_add2, the *source* is added into the *destination*. The source and destination fields must be of the same length.

For CM_add_constant, the *source_constant* (a long integer) is sign-extended or truncated to *length* bits and added into the *destination*.

The carry flag receives the carry out from the high-order bit position. The *length* must be greater than or equal to one.

The overflow flag is set if the sum cannot be represented in the destination field, and is cleared otherwise; it is computed as the XOR of the carry into the high-order bit position and the carry out of the high-order bit position.

CM_add_carry(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

The *source* and the carry flag are together added into the *destination*; that is, the destination receives the three-way sum of the destination, the source, and the old carry flag. The source and destination fields must be of the same length. The carry flag then receives the carry out from the high-order bit position. The *length* must be greater than or equal to one.

The overflow flag is set if the sum cannot be represented in the destination field, and is cleared otherwise; it is computed as the XOR of the carry into the high-order bit position and the carry out of the high-order bit position.

CM_add_flags(source1, source2, length)
```
CM_memaddr_t source1, source2;
unsigned length;
```

The two source fields are added together. These fields must be of the same length. The *length* argument must be greater than or equal to one.

The carry flag and overflow flag are set as for the CM_add2 function, but the sum itself is not written into memory. Only the flags are affected.

CM_add(dest, source1, source2, dlen, slen1, slen2)
```
CM_memaddr_t dest, source1, source2;
unsigned dlen, slen1, slen2;
```

The two source fields are added together and the sum is stored into *dest*. The length of the destination field is specified by *dlen*, the length of the *source1* field is specified by *slen1*, and the length of the *source2* field is specified by *slen2*. The lengths must be greater than or equal to one.

The overflow flag is set if the sum cannot be represented in the destination field, and is cleared otherwise.

If you wish to use only two addresses, *dest* may point to the same field as *source1*. No other overlapping combination is correct.

This function differs from CM_add2 only in allowing independent specification of the destination and first source, in allowing independent specification of the lengths of all three fields, and in not affecting the carry flag.

CM_subtract2(destination, source, length)

```
CM_memaddr_t destination, source;
unsigned length;
```

CM_subtract_constant(destination, source_constant, length)

```
CM_memaddr_t destination;
long source_constant;
unsigned length;
```

For CM_subtract2 the *source* is subtracted from the *destination*. The source and destination fields must be of the same length.

For CM_subtract_constant the *source_constant* (a long integer) is subtracted from the destination. If the *length* is less than 32, the low-order bits of the constant are used.

The carry flag then receives the carry out from the high-order bit position; "carry" is equivalent to "not borrow." The length must be greater than or equal to one.

The overflow flag is set if the difference cannot be represented in the destination field, and is cleared otherwise.

CM_subtract_borrow(destination, source, length)

```
CM_memaddr_t destination, source;
unsigned length;
```

The *source* and the inverse of the carry flag are together subtracted from the *destination*. The carry flag then receives the carry out from the high-order bit position; "carry" is equivalent to "not borrow." The *length* must be greater than or equal to one.

The overflow flag is set if the difference cannot be represented in the destination field, and is cleared otherwise.

CM_subtract(dest, source1, source2, dlen, slen1, slen2)

```
CM_memaddr_t dest, source1, source2;
unsigned dlen, slen1, slen2;
```

The *source2* field is subtracted from the *source1* field, and the difference is stored into *dest*. The length of the destination field is specified by *dlen*, the length of *source1* is specified by *slen1*, and the length of *source2* is specified by *slen2*. These lengths must be greater than or equal to one.

The overflow flag is set if the sum cannot be represented in the destination field, and is cleared otherwise.

This function differs from CM_subtract2 only in allowing independent specification of the destination and first source, in allowing independent specification of the lengths of all three fields, and in not affecting the carry flag.

CM_multiply2(destination, source, length)

```
CM_memaddr_t destination, source;
unsigned length;
```

The *source* is multiplied by the *destination*. The overflow flag is set if the product cannot be represented in the destination field, and is cleared otherwise.

The *length* must be not greater than the value of CM_maximum_integer_length.

CM_multiply(dest, source1, source2, dlen, slen1, slen2)
```
CM_memaddr_t dest, source1, source2;
unsigned dlen, slen1, slen2;
```

The two source fields are multiplied together and the product is stored into *dest*. The length of the destination field is specified by *dlen*, the length of *source1* is specified by *slen1*, and the length of *source2* is specified by *slen2*.

The overflow flag is set if the product cannot be represented in the destination field, and is cleared otherwise.

The lengths *slen1* and *slen2* must each be not greater than the value of `CM_maximum_integer_length`.

This function differs from `CM_multiply2` only in allowing independent specification of the destination and first source and independent specification of the lengths of all three fields.

CM_floor_divide(dest, source1, source2, dlen, slen1, slen2)
```
CM_memaddr_t dest, source1, source2;
unsigned dlen, slen1, slen2;
```

CM_ceiling_divide(dest, source1, source2, dlen, slen1, slen2)
```
CM_memaddr_t dest, source1, source2;
unsigned dlen, slen1, slen2;
```

CM_truncate_divide(dest, source1, source2, dlen, slen1, slen2)
```
CM_memaddr_t dest, source1, source2;
unsigned dlen, slen1, slen2;
```

CM_round_divide(dest, source1, source2, dlen, slen1, slen2)
```
CM_memaddr_t dest, source1, source2;
unsigned dlen, slen1, slen2;
```

The *source1* field is divided by the *source2* field, and the quotient is stored into *dest*. The four functions differ only in the method of rounding when the true mathematical quotient is not an exact integer:

floor rounds toward –infinity.
ceiling rounds toward +infinity.
truncate rounds toward 0.
round rounds to the nearest integer, and if the mathematical quotient lies exactly halfway between two integers, then the even integer is used.

The length of *dest* is specified by *dlen*, the length of *source1* is specified by *slen1*, and the length of *source2* is specified by *slen2*. The lengths *slen1* and *slen2* must each be not greater than the value of `CM_maximum_integer_length`.

The overflow flag is set if the quotient cannot be represented in the destination field, and is cleared otherwise. If *source2* is zero, then the test flag is set and the destination field will contain unpredictable results; otherwise the test flag is cleared.

CM_mod(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

CM_rem(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

The *destination* field (call its value x) is divided by the *source* field (y), producing a quotient (q) and a remainder (r); the quotient is discarded, and the remainder is stored back into the *destination* field.

For CM_mod, the quotient is computed the same way as for CM_floor_divide; CM_rem makes the same computation as for CM_truncate_divide. The remainder is then always computed so as to satisfy the equation $x = q \cdot y + r$.

The *length* must be not greater than the value of CM_maximum_integer_length.

Note that overflow cannot occur. If the source is zero, then the test flag is set and the destination field will contain unpredictable results; otherwise the test flag is cleared.

CM_floor_and_mod(dest1, dest2, source1, source2, dlen, slen1, slen2)
```
CM_memaddr_t dest1, dest2, source1, source2;
unsigned dlen, slen1, slen2;
```

CM_ceiling_and_remainder(dest1, dest2, source1, source2, dlen, slen1, slen2)
```
CM_memaddr_t dest1, dest2, source1, source2;
unsigned dlen, slen1, slen2;
```

CM_truncate_and_rem(dest1, dest2, source1, source2, dlen, slen1, slen2)
```
CM_memaddr_t dest1, dest2, source1, source2;
unsigned dlen, slen1, slen2;
```

CM_round_and_remainder(dest1, dest2, source1, source2, dlen, slen1, slen2)
```
CM_memaddr_t dest1, dest2, source1, source2;
unsigned dlen, slen1, slen2;
```

The *source1* field (call its value x) is divided by the *source2* field (y); the quotient (q) is stored into *dest1*, and the remainder (r) is stored into *dest2*. The quotient is computed the same way as for CM_floor_divide, CM_ceiling_divide, CM_truncate_divide, or CM_round_divide, respectively. The remainder is then always computed so as to satisfy the equation $x = q \cdot y + r$.

The length of *dest1* is specified by *dlen*, the length of *source1* is specified by *slen1*, and the length of *dest2* and *source2* is specified by *slen2*. The lengths *slen1* and *slen2* must each be not greater than the value of CM_maximum_integer_length.

The overflow flag is set if the quotient cannot be represented in the *dest1* field, and is cleared otherwise. Because *dest2* is always the same size as *source2*, the remainder cannot overflow the *dest2* field. If *source2* is zero, then the test flag is set and the destination fields will contain unpredictable results; otherwise the test flag is cleared.

CM_max(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

CM_min(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

The *source* and *destination* fields are considered to be signed integers. If the source of CM_max is greater than the destination, or if the source of CM_min is less, then the source is copied to the destination and the test flag is set; otherwise the test flag is cleared and the destination is unchanged.

CM_max_constant(destination, source_constant, length)
```
CM_memaddr_t destination;
long source_constant;
unsigned length;
```

CM_min_constant(destination, source_constant, length)
```
CM_memaddr_t destination;
long source_constant;
unsigned length;
```

The *destination* field is considered to be a signed integer. If the *source_constant* of CM_max_constant is greater than the destination, or if the *source_constant* of CM_min_constant is less, then the *source_constant* is is sign-extended or truncated to *length* bits and copied to the destination, and the test flag is set. Otherwise the test flag is cleared and the destination is unchanged.

CM_eq(source1, source2, length)
```
CM_memaddr_t source1, source2;
unsigned length;
```

CM_ne(source1, source2, length)
```
CM_memaddr_t source1, source2;
unsigned length;
```

CM_lt(source1, source2, length)
```
CM_memaddr_t source1, source2;
unsigned length;
```

CM_le(source1, source2, length)
```
CM_memaddr_t source1, source2;
unsigned length;
```

CM_gt(source1, source2, length)
```
CM_memaddr_t source1, source2;
unsigned length;
```

CM_ge(source1, source2, length)
```
CM_memaddr_t source1, source2;
unsigned length;
```

The *source1* field is compared with the *source2* field; both are regarded as signed integers. The test flag is set if *source1* is equal to (CM_eq), not equal to (CM_ne), less than (CM_lt), less than or equal to (CM_le), greater than (CM_gt), or greater than or equal to (CM_ge) *source2,* and is cle ıred otherwise.

CM_eq_constant(source, source_constant, length)
```
CM_memaddr_t source;
long source_constant;
unsigned length;
```

CM_ne_constant(source, source_constant, length)
```
CM_memaddr_t source;
long source_constant;
unsigned length;
```

CM_lt_constant(source, source_constant, length)
```
CM_memaddr_t source;
long source_constant;
unsigned length;
```

CM_le_constant(source, source_constant, length)
```
CM_memaddr_t source;
long source_constant;
unsigned length;
```

CM_gt_constant(source, source_constant, length)
```
CM_memaddr_t source;
long source_constant;
unsigned length;
```

CM_ge_constant(source, source_constant, length)
```
CM_memaddr_t source;
long source_constant;
unsigned length;
```

The *source* is compared with the *source_constant*; both are regarded as signed integers. The *source_constant* is sign-extended or truncated to the source *length*. The test flag is set if the source field is equal to (CM_eq_constant), not equal to (CM_ne_constant), less than (CM_lt_constant), less than or equal to (CM_le_constant), greater than (CM_gt_constant), or greater than or equal to (CM_ge_constant) the *source_constant* (a long integer), and is cleared otherwise.

CM_compare(dest, source1, source2, dlen, slen1, slen2)
```
CM_memaddr_t dest, source1, source2;
unsigned dlen, slen1, slen2;
```

The two source fields are compared, and one of the values −1, 0, or 1 is stored into *dest* according to whether *source1* is less than, equal to, or greater than *source2*, respectively. The length of the destination field must be at least 2.

The length of the destination field is specified by *dlen*, the length of *source1* is specified by *slen1*, and the length of *source2* is specified by *slen2*.

CM_shift(destination, source, dlen, slen)
```
CM_memaddr_t destination, source;
unsigned dlen, slen;
```

The contents of the *destination* field are shifted within that field by a distance determined by the *source* field (a signed integer). In this way the destination field can be shifted by a different amount in each selected processor.

If the source field is positive, the shift is to the left (toward the most significant end of the field), and vacated low-order positions are cleared. The overflow flag is set if any bit that differs from the sign bit is shifted into or past the sign bit, and otherwise is cleared.

If the source field is negative, the shift is to the right (toward the least significant end of the field), and vacated high-order positions are filled with copies of the original sign bit of the destination field. The overflow flag is always cleared in this case.

If the source field is zero, then the destination field is unaffected. The overflow flag is always cleared in this case.

Binary Operations on Unsigned Integers

The functions in this section parallel exactly those in the previous section, but operate on unsigned integers rather than signed integers. The descriptions do differ in minor ways, such as in observations about overflow, because of the difference in representation.

CM_u_add2(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

CM_u_add_constant(destination, source_constant, length)
```
CM_memaddr_t destination;
unsigned long source_constant;
unsigned length;
```

For CM_u_add2, the *source* is added into the *destination*. The source and destination fields must be of the same length.

For CM_u_add_constant, the *source_constant* (an unsigned long integer) is added into the destination. The *source_constant* is zero-extended or truncated to the destination *length*.

The carry flag receives the carry out from the high-order bit position. The *length* must be greater than or equal to one.

The overflow flag is set if the sum cannot be represented in the destination field, and is cleared otherwise; it is computed as the carry out of the high-order bit position.

CM_u_add_carry(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

The *destination* receives the three-way sum of the *destination*, the *source*, and the old carry flag. The source and destination fields must be of the same length. The carry flag then receives the carry out from the high-order bit position. The *length* must be greater than or equal to one.

The overflow flag is set if the sum cannot be represented in the destination field, and is cleared otherwise; it is computed as the carry out of the high-order bit position.

CM_u_add_flags(source1, source2, length)
```
CM_memaddr_t source1, source2;
unsigned length;
```

The two sources are added together as unsigned integers. The source fields must be of the same length. The *length* must be greater than or equal to one.

The carry flag and overflow flag are set as for the CM_u_add2 function, but the sum itself is not written into memory. Only the flags are affected.

CM_u_add(dest, source1, source2, dlen, slen1, slen2)
```
CM_memaddr_t dest, source1, source2;
unsigned dlen, slen1, slen2;
```

The two source fields are added together and the sum is stored into *dest*. The length of the destination field is specified by *dlen*, the length of *source1* is specified by *slen1*, and the length of *source2* is specified by *slen2*. These lengths must be greater than or equal to one.

The overflow flag is set if the sum cannot be represented in the destination field, and is cleared otherwise.

This function differs from CM_u_add2 only in allowing independent specification of the destination and first source, in allowing independent specification of the lengths of all three fields, and in not affecting the carry flag.

CM_u_subtract2(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

CM_u_subtract_constant(destination, source_constant, length)
```
CM_memaddr_t destination;
unsigned long source_constant;
unsigned length;
```

For CM_u_subtract2, the *source* is subtracted from the *destination*. The source and destination fields must be of the same length.

For CM_u_subtract_constant, the *source_constant* (an unsigned long integer) is subtracted from the destination. The *source_constant* is zero-extended or truncated to the destination *length*. The *length* must be greater than or equal to one.

The carry flag then receives the carry out from the high-order bit position; "carry" is equivalent to "not borrow."

The overflow flag is set if the difference cannot be represented in the destination field, and is cleared otherwise.

CM_u_subtract_borrow(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

The *source* and the inverse of the carry flag are together subtracted from the *destination*. The carry flag then receives the carry out from the high-order bit position; "carry" is equivalent to "not borrow." The *length* must be greater than or equal to one.

The overflow flag is set if the difference cannot be represented in the destination field, and is cleared otherwise.

CM_u_subtract(dest, source1, source2, dlen, slen1, slen2)
```
CM_memaddr_t dest, source1, source2;
unsigned dlen, slen1, slen2;
```

The *source2* field is subtracted from the *source1* field, and the difference is stored into *dest*. The length of *dest* is specified by *dlen*, the length of *source1* is specified by *slen1*, and the length of *source2* is specified by *slen2*. These lengths must be greater than or equal to one.

The overflow flag is set if the sum cannot be represented in the destination field, and is cleared otherwise.

This function differs from CM_u_subtract2 only in allowing independent specification of the destination and first source, in allowing independent specification of the lengths of all three fields, and in not affecting the carry flag.

CM_u_multiply2(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

The *source* is multiplied by the *destination*. The overflow flag is set if the product cannot be represented in the destination field, and is cleared otherwise.

CM_u_multiply(dest, source1, source2, dlen, slen1, slen2)
```
CM_memaddr_t dest, source1, source2;
unsigned dlen, slen1, slen2;
```

The two source fields are multiplied together and the product is stored into *dest*. The length of *dest* is specified by *dlen*, the length of *source1* is specified by *slen1*, and the length of *source2* is specified by *slen2*.

The overflow flag is set if the product cannot be represented in the destination field, and is cleared otherwise.

This function differs from `CM_u_multiply2` only in allowing independent specification of the destination and first source and independent specification of the lengths of all three fields.

CM_u_floor_divide(dest, source1, source2, dlen, slen1, slen2)
```
CM_memaddr_t dest, source1, source2;
unsigned dlen, slen1, slen2;
```

CM_u_ceiling_divide(dest, source1, source2, dlen, slen1, slen2)
```
CM_memaddr_t dest, source1, source2;
unsigned dlen, slen1, slen2;
```

CM_u_truncate_divide(dest, source1, source2, dlen, slen1, slen2)
```
CM_memaddr_t dest, source1, source2;
unsigned dlen, slen1, slen2;
```

CM_u_round_divide(dest, source1, source2, dlen, slen1, slen2)
```
CM_memaddr_t dest, source1, source2;
unsigned dlen, slen1, slen2;
```

The *source1* field is divided by the *source2* field, and the quotient is stored into *dest*. The four functions differ only in the method of rounding when the true mathematical quotient is not an exact integer:

floor rounds toward –infinity.
ceiling rounds toward +infinity.
truncate rounds toward 0.
round rounds to the nearest integer, and if the mathematical quotient lies exactly halfway between two integers, then the even integer is used.

Note that `CM_u_floor_divide` and `CM_u_truncate_divide` have identical behavior because the operands are unsigned and therefore are always positive.

The length of *dest* is specified by *dlen*, the length of *source1* is specified by *slen1*, and the length of *source2* is specified by *slen2*. The lengths *slen1* and *slen2* must each be not greater than the value of `CM_maximum_integer_length`.

The overflow flag is set if the quotient cannot be represented in the destination field, and is cleared otherwise. If *source2* is zero, then the test flag is set and the destination field will contain unpredictable results; otherwise the test flag is cleared.

CM_u_mod(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

CM_u_rem(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

The *destination* field (call its value x) is divided by the *source* field (y), producing a quotient (q) and a remainder (r); the quotient is discarded, and the remainder is stored back in the destination field.

For CM_u_mod, the quotient is computed the same way as for CM_u_floor_divide; CM_u_rem makes the same computation as for CM_u_truncate_divide. The remainder is then always computed so as to satisfy the equation $x = q \cdot y + r$.

Because the operands are unsigned numbers, CM_u_mod and CM_u_rem have identical behavior.

The *length* must be not greater than the value of CM_maximum_integer_length.

Note that overflow cannot occur. If the source is zero, then the test flag is set and the destination field will contain unpredictable results; otherwise the test flag is cleared.

CM_u_floor_and_mod(dest1, dest2, source1, source2, dlen, slen1, slen2)
```
CM_memaddr_t dest1, dest2, source1, source2;
unsigned dlen, slen1, slen2;
```

CM_u_ceiling_and_remainder(dest1, dest2, source1, source2, dlen, slen1, slen2)
```
CM_memaddr_t dest1, dest2, source1, source2;
unsigned dlen, slen1, slen2;
```

CM_u_truncate_and_rem(dest1, dest2, source1, source2, dlen, slen1, slen2)
```
CM_memaddr_t dest1, dest2, source1, source2;
unsigned dlen, slen1, slen2;
```

CM_u_round_and_remainder(dest1, dest2, source1, source2, dlen, slen1, slen2)
```
CM_memaddr_t dest1, dest2, source1, source2;
unsigned dlen, slen1, slen2;
```

The *source1* field (call its value x) is divided by the *source2* field (y); the quotient (q) is stored in the *dest1* field, and the remainder (r) is stored in the *dest2* field. The quotient is computed the same way as for CM_u_floor_divide, CM_u_ceiling_divide, CM_u_truncate_divide, or CM_u_round_divide, respectively. The remainder is then always computed so as to satisfy the equation $x = q \cdot y + r$.

The length of *dest1* is specified by *dlen*, the length of *source1* is specified by *slen1*, and the length of *dest2* and *source2* is specified by *slen2*. The lengths *slen1* and *slen2* must each be not greater than the value of CM_maximum_integer_length.

Because the operands are unsigned numbers, the functions CM_u_floor_and_mod and CM_u_truncate_and_rem have identical behavior.

The overflow flag is set if the quotient cannot be represented in the *dest1* field, and is cleared otherwise. Because *dest2* is always the same size as *source2*, the remainder cannot overflow the *dest2* field. If *source2* is zero, then the test flag is set and the destination fields will contain unpredictable results; otherwise the test flag is cleared.

CM_u_max(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

CM_u_min(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

The *source* and *destination* fields are regarded as unsigned integers. If the source is greater (CM_u_max) or less (CM_u_min) than the destination, then the source is copied to the destination and the test flag is set; otherwise the test flag is cleared and the destination is unchanged.

CM_u_max_constant(destination, source_constant, length)
```
CM_memaddr_t destination;
unsigned long source_constant;
unsigned length;
```

CM_u_min_constant(destination, source_constant, length)
```
CM_memaddr_t destination;
unsigned long source_constant;
unsigned length;
```

The *destination* field is regarded as an unsigned integer. The *source_constant* is zero-extended or truncated to the destination *length*. If the *source_constant* is greater (CM_u_max_constant) or less (CM_u_min_constant) than the destination, then the *source_constant* is copied to the destination and the test flag is set; otherwise the test flag is cleared and the destination is unchanged.

CM_u_eq(source1, source2, length)
```
CM_memaddr_t source1, source2;
unsigned length;
```

CM_u_ne(source1, source2, length)
```
CM_memaddr_t source1, source2;
unsigned length;
```

CM_u_lt(source1, source2, length)
```
CM_memaddr_t source1, source2;
unsigned length;
```

CM_u_le(source1, source2, length)
```
CM_memaddr_t source1, source2;
unsigned length;
```

CM_u_gt(source1, source2, length)
```
CM_memaddr_t source1, source2;
unsigned length;
```

CM_u_ge(source1, source2, length)
```
CM_memaddr_t source1, source2;
unsigned length;
```

The *source1* field is compared with the *source2* field; both are regarded as unsigned integers. The test flag is set if *source1* is equal to (CM_u_eq), not equal to (CM_u_ne), less than (CM_u_lt), less than or equal to (CM_u_le), greater than (CM_u_gt), or greater than or equal to (CM_u_ge) *source2*, and is cleared otherwise.

Of course, CM_u_eq behaves identically to CM_eq, and CM_u_ne behaves identically to CM_ne. They are provided primarily for reasons of symmetry.

CM_u_eq_constant(source, source_constant, length)
```
CM_memaddr_t source;
 source_constant;
unsigned length;
```

CM_u_ne_constant(source, source_constant, length)
```
CM_memaddr_t source;
unsigned long source_constant;
unsigned length;
```

CM_u_lt_constant(source, source_constant, length)
```
CM_memaddr_t source;
unsigned long source_constant;
unsigned length;
```

CM_u_le_constant(source, source_constant, length)
```
CM_memaddr_t source;
unsigned long source_constant;
unsigned length;
```

CM_u_gt_constant(source, source_constant, length)
```
CM_memaddr_t source;
unsigned long source_constant;
unsigned length;
```

CM_u_ge_constant(source, source_constant, length)
```
CM_memaddr_t source;
unsigned long source_constant;
unsigned length;
```

The *source* is compared with the *source_constant*; both are regarded as unsigned integers. The *source_constant* is zero-extended or truncated to the source *length*. The test flag is set if the source field is equal to (CM_u_eq_constant), not equal to (CM_u_ne_constant), less than (CM_u_lt_constant), less than or equal to (CM_u_le_constant), greater than (CM_u_gt_constant), or greater than or equal to (CM_ge_constant) the *source_constant* (an unsigned long integer), and is cleared otherwise.

Of course, CM_u_eq_constant behaves identically to CM_eq_constant, and CM_u_ne_constant behaves identically to CM_ne_constant. They are provided primarily for reasons of symmetry.

CM_u_compare(dest, source1, source2, dlen, slen1, slen2)
```
CM_memaddr_t dest, source1, source2;
unsigned dlen, slen1, slen2;
```

The two source fields are compared, and one of the values −1, 0, or 1 is stored into *dest* according to whether *source1* is less than, equal to, or greater than *source2*, respectively. The length of the destination field must be at least 2.

The destination is always stored as a signed integer, despite the fact that both source fields are regarded as unsigned integers.

The length of *dest* is specified by *dlen*, the length of *source1* is specified by *slen1*, and the length of *source2* is specified by *slen2*.

CM_u_shift(destination, source, dlen, slen)
```
CM_memaddr_t destination, source;
unsigned dlen, slen;
```

The contents of the *destination* field are shifted within that field by a distance determined by the *source* field (which is a *signed* integer). In this way the destination field can be shifted by a different amount in each selected processor.

If the source field is positive, the shift is to the left (toward the most significant end of the field), and vacated low-order positions are cleared. The overflow flag is set if any bit containing 1 is shifted out, and otherwise is cleared.

If the source field is negative, the shift is to the right (toward the least significant end of the field), and vacated high-order positions are cleared. The overflow flag is always cleared in this case.

If the source field is zero, then the destination field is unaffected. The overflow flag is always cleared in this case.

Binary Operations on Floating-Point Numbers

Following the conventions of IEEE floating-point arithmetic, the overflow flag is "sticky" for floating-point operations; a function may set the flag, but will never clear it. This is different from the behavior for integer arithmetic.

CM_f_add(destination, source, signif_len, expt_len)
```
CM_memaddr_t destination, source;
unsigned signif_len, expt_len;
```

CM_f_subtract(destination, source, signif_len, expt_len)
```
CM_memaddr_t destination, source;
unsigned signif_len, expt_len;
```

CM_f_multiply(destination, source, signif_len, expt_len)
```
CM_memaddr_t destination, source;
unsigned signif_len, expt_len;
```

CM_f_divide(destination, source, signif_len, expt_len)
```
CM_memaddr_t destination, source;
unsigned signif_len, expt_len;
```

These are the standard four arithmetic operations on floating-point numbers.

In each case, the length *signif_len* must be not greater than the value of `CM_maximum_significand_length`, and the length *expt_len* must be not greater than the value of `CM_maximum_exponent_length`.

The result is rounded if necessary to make it fit into a significand of length *signif_len*. (If the mathematically correct result lies exactly between two representable floating-point numbers, then it is rounded to the floating-point value whose significand's least significant bit is zero; this is the "round to even" rule.)

If the correct result cannot be represented because its exponent is too large (overflow) or too small (underflow), then the overflow flag is set and the contents of the destination field are unpredictable. Otherwise the overflow flag is unaffected. The test flag is unaffected by the functions `CM_f_add`, `CM_f_subtract`, and `CM_f_multiply`; for the function `CM_f_divide`, the test flag is set in the case of division by zero, and is unaffected otherwise.

CM_f_max(destination, source, signif_len, expt_len)
```
CM_memaddr_t destination, source;
unsigned signif_len, expt_len;
```

CM_f_min(destination, source, signif_len, expt_len)
```
CM_memaddr_t destination, source;
unsigned signif_len, expt_len;
```

If the *source* is greater (CM_f_max) or less (CM_f_min) than the *destination*, then the source is copied to the destination and the test flag is set; otherwise the test flag is cleared and the destination is unchanged.

CM_f_eq(source1, source2, signif_len, expt_len)
```
CM_memaddr_t source1, source2;
unsigned signif_len, expt_len;
```

CM_f_ne(source1, source2, signif_len, expt_len)
```
CM_memaddr_t source1, source2;
unsigned signif_len, expt_len;
```

CM_f_lt(source1, source2, signif_len, expt_len)
```
CM_memaddr_t source1, source2;
unsigned signif_len, expt_len;
```

CM_f_le(source1, source2, signif_len, expt_len)
```
CM_memaddr_t source1, source2;
unsigned signif_len, expt_len;
```

CM_f_gt(source1, source2, signif_len, expt_len)
```
CM_memaddr_t source1, source2;
unsigned signif_len, expt_len;
```

CM_f_ge(source1, source2, signif_len, expt_len)
```
CM_memaddr_t source1, source2;
unsigned signif_len, expt_len;
```

The two source fields are compared; both are regarded as floating-point numbers. The test flag is set if *source1* is equal to (CM_f_eq), not equal to (CM_f_ne), less than (CM_f_lt), less than or equal to (CM_f_le), greater than (CM_f_lt), or greater than or equal to (CM_f_le) *source2*, and is cleared otherwise.

CM_f_compare(dest, source1, source2, dlen, signif_len, expt_len)
```
CM_memaddr_t dest, source1, source2;
unsigned dlen, signif_len, expt_len;
```

The two source fields are compared, and one of the values −1, 0, or 1 is stored into *dest* according to whether *source1* is less than, equal to, or greater than *source2*, respectively. The length of the destination field must be at least 2.

The destination is always stored as a signed integer, despite the fact that both source fields are regarded as floating-point numbers.

The length of *dest* is specified by *dlen*, the length of *source1* is specified by *slen1*, and the length of *source2* is specified by *slen2*.

Chapter 7
Other Simple Operations

Each function in this chapter operates on one or more fields. The operation is performed in every processor selected by the context flag.

If a flag is not explicitly mentioned in the individual description of a function, then it leaves the flag unaffected.

Movement of Fields

CM_move(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

CM_move_always(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

The *source* is copied into the *destination*. The function CM_move copies conditionally; CM_move_always copies unconditionally.

The move operation is unusual in that it always operates correctly in the face of overlapping fields; it always behaves as if all the bits of the source field were copied to an invisible buffer, and the buffer were then copied into the destination field.

These functions may be used to operate on the flags.

CM_u_move(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

CM_u_move_always(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

The unsigned versions of these move operations are identical in function to the signed versions, and are provided for symmetry.

CM_move_constant(destination, source_constant, length)
```
CM_memaddr_t destination;
long source_constant;
unsigned length;
```

CM_move_constant_always(destination, source_constant, length)
```
CM_memaddr_t destination;
long source_constant;
unsigned length;
```

The *source_constant* is copied into the *destination*. If the *source_constant* is longer than the destination field, the low-order bits are copied; if it is shorter it is sign-extended to *length*.

The function CM_move_constant copies conditionally; CM_move_constant_always copies unconditionally.

An appropriate way to clear a field is to move the constant zero into it:

```
CM_move_constant(destination, 0, length);
```

These functions may be used to operate on the flags. The standard way to turn on the context flag in all processors is:

```
CM_move_constant_always(CM_context_flag, 1, 1);
```

CM_u_move_constant(destination, source_constant, length)

```
CM_memaddr_t destination;
unsigned long source_constant;
unsigned length;
```

CM_u_move_constant_always(destination, source_constant, length)

```
CM_memaddr_t destination;
unsigned long source_constant;
unsigned length;
```

The unsigned versions of these move operations are identical in function to the signed versions, except that when the source constant is shorter in length than the destination, the high-order bits are zeroed.

CM_move_string_constant(destination, source_constant_ptr, length)

```
CM_memaddr_t destination;
char *source_constant_ptr;
unsigned length;
```

CM_move_string_constant_always(destination, source_constant_ptr, length)

```
CM_memaddr_t destination;
char *source_constant_ptr;
unsigned length;
```

These two functions are identical to their unsigned counterparts above, except that *source_constant_ptr* is a pointer to a field whose contents is copied to the destination. The field is zero-extended or truncated to the destination *length*.

CM_f_move(destination, source, signif_len, expt_len)

```
CM_memaddr_t destination, source;
unsigned signif_len, expt_len;
```

The *source* is copied into the *destination* as a floating-point number. This function is unusual in that it always operates correctly in the face of overlapping fields; it always behaves as if all the bits of the source field were copied to an invisible buffer, and the buffer were then copied into the destination field.

This function is no different from CM_move except that it allows specification of the length in terms of significand and exponent lengths. (When IEEE standard floating-point arithmetic is supported, CM_f_move will detect not-a-number (NAN) values, but CM_move will not.)

CM_f_move_constant(destination, source_constant, signif_len, expt_len)
```
CM_memaddr_t destination;
double source_constant;
unsigned signif_len, expt_len;
```

The *source_constant* is copied into the *destination* as a floating-point number.

No indication is given if the value of the *source_constant* cannot be represented in the format specified by *signif_len* and *expt_len*. If the exponent can be represented adequately but the significand cannot, then precision will be lost but the value will be substantially correct (that is, rounding or truncation may occur).

CM_f_move_decoded_constant(destination, signif_constant, expt_constant, sign_constant, signif_len, expt_len)
```
CM_memaddr_t destination;
long signif_constant, expt_constant, sign_constant;
unsigned signif_len, expt_len;
```

The arguments *signif_constant*, *expt_constant*, and *sign_constant* are integers representing a constant floating-point value. This floating-point value is copied into the *destination* field as a floating-point number.

No indication is given if the value of the floating-point constant cannot be represented in the format specified by *signif_len* and *expt_len*. If the exponent can be represented adequately but the significand cannot, then precision will be lost but the value will be substantially correct (that is, rounding or truncation may occur).

CM_move_reversed(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

The *destination* receives the reverse of the *source*. The low-order bit of the source field is copied to the high-order bit of the destination field, the next lowest-order bit of the source field is copied to the next highest-order bit of the destination field, etc. This works when the source and destination fields are the same field, but not for other cases of of overlap.

CM_pop_and_discard(n)
```
unsigned n;
```

The non-negative integer value *n* is added to the stack pointer, thereby popping *n* bits from the stack and discarding them. This function is unconditional.

CM_push_space(n)
```
unsigned n;
```

The non-negative integer value *n* is subtracted from the stack pointer, thereby pushing *n* bits onto the stack without initialization. Stack limit checking is performed. This function is unconditional.

Stack Limit, Pointer, and Upper Bound

The functions for setting the stack limit *l*, pointer *p*, and upper bound *u* enforce the constraint that $0 \leq l \leq p \leq u \leq$ `CM_user_memory_address_limit`.

`CM_memaddr_t`
CM_get_stack_pointer()

The stack pointer, which is the memory address of the topmost (lowest-addressed) bit on the stack, is returned.

CM_set_stack_pointer(place)
`CM_memaddr_t place;`

The stack pointer is set to *place*, which must be a memory address between the stack limit (inclusive) and the stack upper bound (inclusive).

CM_reset_stack_pointer()

The stack pointer is reset so as to empty the stack. It then is equal to the stack upper bound.

`CM_memaddr_t`
CM_get_stack_limit()

The stack limit, which is the memory address of the lowest bit of the gap, is returned.

CM_set_stack_limit(place)
`CM_memaddr_t place;`

The stack limit is set to *place*, which must be a memory address between zero (inclusive) and the stack pointer (inclusive).

`CM_memaddr_t`
CM_get_stack_upper_bound()

The stack upper bound, which is the memory address of the lowest bit of the upper data area (that is, one bit higher than the highest addressed bit of the stack area), is returned.

CM_set_stack_upper_bound(place)
`CM_memaddr_t place;`

The stack upper bound is set to *place*, which must be a memory address between the stack pointer (inclusive) and the value of `CM_user_memory_address_limit` (inclusive).

Arrays

The functions in this section allow each selected processor to treat a portion of its own memory as an array of elements. The element to be fetched or stored is determined by a per-processor index.

CM_array_ref(dest, base, index,
dest_length, index_length, array_length, element_length)
```
CM_memaddr_t dest, base, index;
unsigned dest_length, index_length, array_length, element_length;
```

This is a primitive form of array reference, for arrays stored in the memory of individual processors. Each processor has an array index stored in the field starting at *index*. This is used to index into an array which starts at *base*. The element indexed to, of length *dest_length*, is copied into *dest* in all selected processors. Thus different processors may access different elements of their arrays. More precisely, the field starting at *base* + *i* · *element_length*, where *i* is the unsigned number stored at *index*, is copied to a position starting at *dest*, in all selected processors.

The argument *array_length* is one greater than the largest allowed value of the index. Those processors which have index values greater than or equal to *array_length* will not alter the value of the destination field; they will also clear test flag. All processors in which the index field is less than *array_length* will set test flag. The argument *element_length* is the length of individual elements of the array. Usually this will be the same as *dest_length*, but for certain applications it is worthwhile for it to differ. For example, to extract a 4-character substring from a string of 8-bit characters, *dest_length* is 32 but *element_length* is 8.

This function requires one bit of temporary storage. The size of the gap must be at least this large when the operation is performed, and the contents of the gap may be arbitrarily altered.

CM_array_set(source, base, index,
source_length, index_length, array_length, element_length)
```
CM_memaddr_t source, base, index;
unsigned source_length, index_length, array_length, element_length;
```

This is a primitive form of array modification. Each processor has an array index stored in the field *index*. This is used to index into an array which starts at *base*. The field starting at *source* is copied into this element in all selected processors. More precisely, the field starting at *source*, of length *source_length*, is copied into the field starting at *base* + *i* · *element_length*, where *i* is the unsigned number stored at *index*, in all selected processors.

The argument *array_length* is one greater than the largest allowed value of the index. Those processors which have index values greater than or equal to *array_length* will not transfer any data; they will also clear test flag. All processors in which the index field is less than *array_length* will set test flag. The argument *element_length* is the length of individual elements of the array.

This function requires one bit of temporary storage. The size of the gap must be at least this large when the operation is performed, and the contents of the gap may be arbitrarily altered.

Generating Random Numbers

CM_random(destination, length)
```
CM_memaddr_t destination;
unsigned length;
```

A random value is placed into the *destination* in each selected processor; each selected processor ideally receives an independently chosen random value in the range $-2^{length-1}$ to $2^{length-1} - 1$.

CM_u_random_with_limit(destination, length, limit)
```
CM_memaddr_t destination;
unsigned length, limit;
```

A random unsigned value is placed into the *destination* in each selected processor; each selected processor ideally receives an independently chosen random value less than the *limit*.

CM_initialize_random_number_generator(seed)
```
int seed;
```

The generator of pseudo-random numbers used by the function `CM_random` is initialized. If a seed (a front-end integer) is provided, then that determines the initial state; if no seed is

specified, then a value based on the date and time of day is used. Note that cmcoldboot effectively calls CM_initialize_random_number_generator with no seed.

Controlling the Cabinet Lights

CM_latch_leds(source)
```
CM_memaddr_t source;
```

CM_latch_leds_always(source)
```
CM_memaddr_t source;
```

The specified one-bit field is read from every selected processor (or every processor, for the _always version) and used to determine which lights (LEDs) should be illuminated. There is one light associated with each group of 16 physical processors; each physical processor has some number of virtual processors. Two virtual processors belong to the same group if their virtual processor numbers agree in their twelve most significant bits. A light is illuminated if every selected virtual processor in the group has a zero in the selected *source* field (that is, the fields are combined for each group by a logical NOR operation).

These functions may be used to operate on the flags.

Note that the pattern will actually persist in the lights only if CM_set_system_leds_mode(CM_LEDS_OFF) has been performed; otherwise the Connection Machine system software will present other patterns in the lights.

CM_set_system_leds_mode(mode)
```
integer mode;
```

The lights on the front and back of the Connection Machine system cabinet can be controlled in a variety of ways. The CM_set_system_leds_mode function selects what information will be displayed in the lights. If the specified *mode* is CM_LEDS_OFF, then all the lights are turned off, and thereafter the user functions CM_latch_leds and CM_latch_leds_always may be used to control the lights. (The functions CM_latch_leds and CM_latch_leds_always may still be used when in a system supplied display mode, but the user-specified pattern is unlikely to persist as it may be immediately altered by the system, depending on the mode.)

Chapter 8
Cooperative Computations

Global Operations

Each function in this section operates on one field in every selected processor, combining these fields' values to produce a single result that is then reported back to the front-end computer; that is, each function returns a value and is not just executed for its effect on the Connection Machine memory.

Returning a value to the front end presents the possibility of overflow. Consider first the case of integers. An integer result is always returned in `long` format (signed or unsigned, as appropriate); overflow may occur, but no warning is given. In the case of floating-point computation, overflow may occur in converting the result to a format acceptable to the front end. If overflow occurs the global variable `CM_conversion_overflow` is set.

Global Operations on Bit Fields

```
unsigned long
CM_global_logand(source, length)
CM_memaddr_t source;
unsigned length;

unsigned long
CM_global_logand_always(source, length)
CM_memaddr_t source;
unsigned length;

unsigned long
CM_global_logior(source, length)
CM_memaddr_t source;
unsigned length;

unsigned long
CM_global_logior_always(source, length)
CM_memaddr_t source;
unsigned length;
```

The specified field is examined in all selected processors (for `CM_global_logand` and `CM_global_logior`) or in all processors (for the `_always` versions of the functions). The fields are combined by performing bit-wise logical AND (`CM_global_logand`) or INCLUSIVE OR (`CM_global_logior`) operations. The resulting combined field is then treated as an unsigned integer and returned to the front-end computer. These functions may be used to operate on the flags.

If no processors are selected, then the identity for the operation is returned; this value is $2^{length} - 1$ for `CM_global_logand` and 0 for `CM_global_logior`.

```
long
CM_global_count(source)
CM_memaddr_t source;

long
CM_global_count_always(source)
CM_memaddr_t source;
```

The specified bit is examined in all selected processors (for the function `CM_global_count`) or all processors, regardless of selection (for the function `CM_global_count_always`). The number of bits that contain 1 rather than 0 is returned. This is similar to using `CM_global_u_add` on a one-bit field; however, `CM_global_count` may be used to operate on flags, whereas `CM_global_add` cannot.

If no processors are selected, then 0 is returned.

Global Operations on Signed Integers

```
long
CM_global_add(source, length)
CM_memaddr_t source;
unsigned length;
```

The specified field is examined in all selected processors, and all the field values, regarded as signed integers, are added together and returned.

If overflow occurs, the low-order bits of the result are returned.

If no processors are selected, then 0 is returned.

This function requires *length* + 1 bits of temporary storage. The size of the gap must be at least this large when it is performed, and the contents of the gap may be arbitrarily altered.

```
long
CM_global_max(source, length)
CM_memaddr_t source;
unsigned length;

long
CM_global_min(source, length)
CM_memaddr_t source;
unsigned length;
```

The specified field is examined in all selected processors. If at least one processor is selected, then `CM_global_max` returns the largest of the field values as a signed integer; `CM_global_min` returns the smallest.

The test flag is set in every selected processor whose field is equal to the finally computed value, and is cleared in all other selected processors.

If overflow occurs, the low-order bits of the result are returned.

If no processors are selected, then $-2^{length-1}$ is returned for `CM_global_max`, and $2^{length-1} - 1$ is returned for `CM_global_min`. The global variable `CM_no_processors_selected` is set if no processors were selected, and cleared otherwise.

Global Operations on Unsigned Integers

```
unsigned long
CM_global_u_add(source, length)
CM_memaddr_t source;
unsigned length;
```

The specified field is examined in all selected processors, and all the field values, regarded as unsigned integers, are added together and returned as an unsigned integer.

If overflow occurs, the low-order bits of the result are returned.

If no processors are selected, then 0 is returned.

This function requires *length* + 1 bits of temporary storage. The size of the gap must be at least this large when it is performed, and the contents of the gap may be arbitrarily altered.

```
unsigned long
CM_global_u_max(source, length)
CM_memaddr_t source;
unsigned length;
```

```
unsigned long
CM_global_u_min(source, length)
CM_memaddr_t source;
unsigned length;
```

The specified field is examined in all selected processors. If at least one processor is selected, then `CM_global_max` returns the largest of the field values as an unsigned integer; `CM_global_min` returns the smallest.

The test flag is set in every selected processor whose field is equal to the finally computed value, and is cleared in all other selected processors.

If overflow occurs, the low-order bits of the result are returned.

If no processors are selected, then 0 is returned for `CM_global_u_max`, and $2^{length} - 1$ is returned for `CM_global_u_min`. Theglobal variable `CM_no_processors_selected` is set if no processors were selected, and is cleared otherwise.

Global Operations on Floating-Point Numbers

```
double
CM_global_f_max(source, signif_len, expt_len)
CM_memaddr_t source;
unsigned signif_len, expt_len;
```

```
double
CM_global_f_min(source, signif_len, expt_len)
CM_memaddr_t source;
unsigned signif_len, expt_len;
```

The specified field is examined in all selected processors. If at least one processor is selected, then `CM_global_max` returns the largest of the field values as floating-point numbers; `CM_global_min` returns the smallest. The result coming from the Connection Machine system is lengthened or shortened as necessary to accommodate the front-end machine's floating-point format. The result is always of type `double`. The global variable `CM_no_processors_selected` is set if no processors were selected, and is cleared otherwise.

Overflow cannot occur during the main calculation, but it may occur in reducing the result to fit the format of the front-end machine.

If no processors are selected, then a very large negative value is returned for `CM_global_max`, and a very large positive value is returned for `CM_global_min`. (When IEEE floating-point arithmetic is implemented, these values will be minus infinity and plus infinity, respectively.) Theglobal variable `CM_no_processors_selected` is set if no processors were selected, and cleared otherwise.

Enumeration

CM_max_scan(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

CM_u_max_scan(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

CM_f_max_scan(destination, source, signif_len, expt_len)
```
CM_memaddr_t destination, source;
unsigned length;
```

Within each selected processor (call its virtual cube address *a*), the *destination* receives the maximum of the *source* fields (regarded as signed integers, unsigned integers, or floating-point numbers, respectively) of all selected processors whose virtual cube address is less than or equal to *a*. In other words, these are max-prefix operations over the set (ordered by cube address) of selected processors.

For example, suppose that processors 3, 4, 7, and 9 are selected and no others, and their source fields respectively contain the values 6, 2, 8, 5. A call to `CM_u_max_scan` would produce the results 6, 6, 8, and 8 in the respective destination fields.

The functions `CM_max_scan` and `CM_u_max_scan` each require 135 bits of temporary storage if *length* is greater than 64, and otherwise require no temporary storage. The function `CM_f_max_scan` requires 136 bits of temporary storage if *length* is greater than 64, and otherwise requires one bit of temporary storage. The gap must be at least as large as required by these functions when they are performed, and the contents of the gap may be arbitrarily altered.

CM_add_scan(destination, source, dlen, slen)
```
CM_memaddr_t destination, source;
unsigned dlen, slen;
```

CM_u_add_scan(destination, source, dlen, slen)
```
CM_memaddr_t destination, source;
unsigned dlen, slen;
```

Within each selected processor (call its virtual cube address *a*), the *destination* receives the sum (signed or unsigned, respectively) of the *source* fields of all selected processors whose virtual cube address is less than or equal to *a*. In other words, these are sum-prefix operations over the set (ordered by cube address) of selected processors.

For example, suppose that processors 3, 4, 7, and 9 are selected and no others, and that each of these processors happens to have its own cube address in the source field. A call to

CM_u_add_scan would produce the results 3, 7, 14, and 25, in the respective destination fields.

To avoid overflow, the following arbitrary restriction is imposed: the length of the destination, *dlen*, must be at least as great as the sum of *slen* and the value of CM_cube_address_length.

These functions require *slen* + *dlen* + 2*c* + 3 bits of temporary storage, where *c* is the value of CM_cube_address_length. The size of the gap must be at least this large when they are performed, and the contents of the gap may be arbitrarily altered.

CM_enumerate(destination, length)
```
CM_memaddr_t destination;
unsigned length;
```

CM_enumerate_and_count(destination, length)
```
CM_memaddr_t destination;
unsigned length;
```

If n processors are selected, a unique unsigned integer between 0 and $n - 1$, inclusive, is stored in the *destination* of each selected processor. The integers are assigned according to the ordering of virtual cube addresses; that is, all the selected processors are numbered in order and this order is determined by their virtual cube addresses. If the processor with virtual cube address p_1 is assigned integer k_1, and the processor with virtual cube address p_2 is assigned integer k_2, then $k_1 < k_2$ if and only if $p_1 < p_2$. This implies that selecting the same set of processors will always produce the same enumeration, and a variety of other useful properties.

The overflow flag is set if the integer for a given processor cannot be represented as an unsigned integer of the specified *length*.

The function CM_enumerate_and_count not only performs the enumeration, but also returns to the front end, as an unsigned integer, the number of selected processors, as if CM_global_count(CM_context_flag) had been performed. The function CM_enumerate simply performs the enumeration and returns no useful value.

Sorting

CM_rank(destination, key, destination_length, key_length)
```
CM_memaddr_t destination, key;
unsigned destination_length, key_length;
```

CM_u_rank(destination, key, destination_length, key_length)
```
CM_memaddr_t destination, key;
unsigned destination_length, key_length;
```

CM_f_rank(destination, key, destination_length, key_signif_len, key_expt_len)
```
CM_memaddr_t destination, key;
unsigned destination_length, key_signif_len, key_expt_len;
```

The ranking operation determines the ordering necessary to sort the *key* fields of the selected processors. It does not actually move data, but merely indicates where the data should be moved to sort it. The *destination* in each selected processor receives, as an unsigned integer, the rank of that processor's key within the set of all selected keys. The smallest key has rank 0, the next smallest has rank 1, and so on; the largest key has rank $n - 1$ where n is the number of selected processors.

The following sequence would cause the data to actually be sorted:

```
CM_rank(rank, key, ranklen, keylen);
CM_send(result, rank, key, keylen, CM_NOLIMITS);
```

The data originally at location key within the selected processors would to be placed, in sorted order, at location `result` within processors 0 through $n - 1$ (the cube addresses given by `rank`). The `CM_rank` function would determine the ranking, and the `CM_send` function would use this ranking to re-order the data in memory. An advantage of uncoupling the rank determination from the actual re-ordering is that the data to be moved may be much larger than the key determining the order, and indeed it may be desirable to re-order other data but not the key itself. In this way ranking and re-ordering each need operate only on the relevant data.

The `CM_rank` function regards the *key* argument as a signed integer, `CM_u_rank` regards it as an unsigned integer, and `CM_f_rank` regards it as a floating-point number.

The *destination_length* must be at least as large as the base two logarithm of the value of `CM_user_number_of_processors_limit`.

The *key_length* must not be greater than $m - c - 1$, where m is the value of `CM_maximum_message_length` and c is the value of `CM_cube_address_length`.

These functions each require $2c$ + *key_length* + 3 bits of temporary storage, where c is the value of `CM_cube_address_length`. The size of the gap must be at least this large when they are performed, and the contents of the gap may be arbitrarily altered.

Chapter 9
Interprocessor Communication

General Communication through the Router

The communications functions in this section transmit data in a general fashion that effectively allows any processor to communicate directly with any other processor.

CM_send_with_overwrite(destination, cube_address, source, length, time_limit)
```
CM_memaddr_t destination, cube_address, source;
unsigned length;
integer time_limit;
```

CM_send_with_logior(destination, cube_address, source, length, time_limit)
```
CM_memaddr_t destination, cube_address, source;
unsigned length;
integer time_limit;
```

CM_send_with_logand(destination, cube_address, source, length, time_limit)
```
CM_memaddr_t destination, cube_address, source;
unsigned length;
integer time_limit;
```

CM_send_with_logxor(destination, cube_address, source, length, time_limit)
```
CM_memaddr_t destination, cube_address, source;
unsigned length;
integer time_limit;
```

CM_send_with_add(destination, cube_address, source, length, time_limit)
```
CM_memaddr_t destination, cube_address, source;
unsigned length;
integer time_limit;
```

CM_send_with_max(destination, cube_address, source, length, time_limit)
```
CM_memaddr_t destination, cube_address, source;
unsigned length;
integer time_limit;
```

CM_send_with_min(destination, cube_address, source, length, time_limit)
```
CM_memaddr_t destination, cube_address, source;
unsigned length;
integer time_limit;
```

CM_send_with_u_max(destination, cube_address, source, length, time_limit)
```
CM_memaddr_t destination, cube_address, source;
unsigned length;
integer time_limit;
```

CM_send_with_u_min(destination, cube_address, source, length, time_limit)
```
CM_memaddr_t destination, cube_address, source;
unsigned length;
integer time_limit;
```

CM_send(destination, cube_address, source, length, time_limit)
```
CM_memaddr_t destination, cube_address, source;
unsigned length;
integer time_limit;
```

For every selected processor p_S, a message *length* bits long is sent from that processor to the processor p_d whose absolute cube address is stored at location *cube_address* in the memory of processor p_S. The message is taken from the field starting at location *source* within processor p_S and is stored into the field at location *destination* within processor p_d.

As an example, one might write:

```
CM_push_space(CM_cube_address_length + 32);
CM_my_cube_address(CM_stack(32));
CM_subtract_constant(CM_stack(32), 1, CM_cube_address_length);
CM_move(CM_stack(0), this_value, 32);
CM_multiply2(CM_stack(0), that_value, 32);
CM_send_with_overwrite(43, CM_stack(32), CM_stack(0), 32, CM_NOLIMIT);
CM_pop_and_discard(CM_cube_address_length + 32);
```

to cause every selected processor to send the product of the 32-bit quantities at addresses `this_value` and `that_value` to the 32-bit field at location 43 in the processor whose cube address is one less than that of the sender.

This operation is conditional, but whether a message is sent depends only on the context flag of the originating processor p_S; the message, once transmitted to processor p_d, is stored regardless of the context flag of processor p_d.

The *length* must be not greater than the value of `CM_maximum_message_length`.

If more than one message is sent to a given processor, then the functions differ in their treatment of collisions.

The `CM_send_with_overwrite` function will store one of the messages sent and will discard all the others without notice.

The `CM_send_with_logand` function will combine all messages and the original contents of the destination field with a bit-wise logical AND operation. To receive the logical AND of only the messages, the destination area should first be set to all ones in all processors that might receive a message.

The `CM_send_with_logior` function combines incoming messages with a bit-wise logical INCLUSIVE OR operation. To receive the logical INCLUSIVE OR of only the messages, the destination area should first be cleared in all processors that might receive a message.

The `CM_send_with_logxor` function combines incoming messages with a bit-wise logical EXCLUSIVE OR operation. To receive the logical EXCLUSIVE OR of only the messages, the destination area should first be cleared in all processors that might receive a message.

The `CM_send_with_add` function adds incoming messages together. The overflow flag is set in a destination processor if overflow occurs at any step. All other processors have the overflow flag cleared unconditionally. The carry flag is not affected. To receive the sum of only the messages, the destination area should first be cleared in all processors that might receive a message.

The `CM_send_with_max`, `CM_send_with_min`, `CM_send_with_unsigned_max`, and `CM_send_with_u_min` functions are similar, but combine incoming messages using signed or

unsigned max or min operations. The test flag is not affected by the max and min operations, but rather is set as described below. When doing a CM_send_with_max or CM_send_with_u_max, to receive the maximum of only the messages, the destination area should first be set to the minimum possible number of the appropriate type and length in all processors that might receive a message. When doing a CM_send_with_min or CM_send_with_u_min, to receive the minimum of only the messages, the destination area should first be set to the maximum possible number of the appropriate type and length in all processors that might receive a message.

The CM_send function is similar, but combines incoming messages in an unpredictable manner. It may be used when the programmer can guarantee that no processor will receive more than one message. Using CM_send when it is appropriate may speed message delivery. The destination area need not be prepared.

For any of these sending operations, every virtual processor that receives one or more messages will have its test flag unconditionally set to 1. Every processor that receives no messages will have its test flag cleared.

If the optional argument *time_limit* is non-negative, the delivery operation will be terminated at a fixed time, rather than being allowed to run until completion. The value of *time_limit* determines the time that will be allowed for messages to be delivered. If the time limit is too short to deliver all the messages, some will not arrive at their destinations. It will be as though those messages were never sent. Which messages will be delivered is unpredictable in any given case, but certain properties are guaranteed. If *time_limit* is 0, no messages will be delivered. For some sufficiently large value of *time_limit*, all messages will be delivered; this value depends on the distribution of selected processors, and the contents of their *cube_address* fields. Two identical calls to a sending function, with identical patterns of selected processors and absolute cube addresses, will deliver exactly the same set of messages. For any distribution of selected processors and absolute cube addresses, increasing *time_limit* will not decrease the set of messages which are delivered. For aesthetic reasons, the symbol CM_NOLIMITS should be used in place of a negative number to indicate the absence of a time limit.

CM_store_with_overwrite(memory_address, source, length, time_limit)
```
CM_memaddr_t memory_address, source;
unsigned length;
integer time_limit;
```

CM_store_with_logior(memory_address, source, length, time_limit)
```
CM_memaddr_t memory_address, source;
unsigned length;
integer time_limit;
```

CM_store_with_logand(memory_address, source, length, time_limit)
```
CM_memaddr_t memory_address, source;
unsigned length;
integer time_limit;
```

CM_store_with_logxor(memory_address, source, length, time_limit)
```
CM_memaddr_t memory_address, source;
unsigned length;
integer time_limit;
```

CM_store_with_add(memory_address, source, length, time_limit)
```
CM_memaddr_t memory_address, source;
unsigned length;
integer time_limit;
```

CM_store_with_max(memory_address, source, length, time_limit)
```
CM_memaddr_t memory_address, source;
unsigned length;
integer time_limit;
```

CM_store_with_min(memory_address, source, length, time_limit)
```
CM_memaddr_t memory_address, source;
unsigned length;
integer time_limit;
```

CM_store_with_u_max(memory_address, source, length, time_limit)
```
CM_memaddr_t memory_address, source;
unsigned length;
integer time_limit;
```

CM_store_with_u_min(memory_address, source, length, time_limit)
```
CM_memaddr_t memory_address, source;
unsigned length;
integer time_limit;
```

CM_store(memory_address, source, length, time_limit)
```
CM_memaddr_t memory_address, source;
unsigned length;
integer time_limit;
```

For every selected processor p_S, a message *length* bits long is sent to a destination specified by the absolute virtual memory address stored at location *memory_address* in the memory of processor p_S. This virtual memory address specifies a virtual processor p_d and a bit address b_d within that processor. The virtual memory address field consists of the absolute cube address of the destination processor followed by the memory within that processor. The message is taken from the field starting at location *source* within processor p_S and is stored into the field at location b_d within processor p_d.

This operation is conditional, but whether a message is sent depends only on the context flag of the originating processor p_S; the message, once transmitted to processor p_d, is stored regardless of the context flag of processor p_d.

The *length* must be not greater than the value of `CM_maximum_message_length`.

If more than one message is sent to a given processor, then the functions differ in their treatment of collisions. In this respect the `CM_store_...` functions are entirely analogous to the corresponding `CM_send_...` functions described above.

For any of these storing operations, every virtual processor that receives one or more messages will have its test flag unconditionally set to 1. Every processor that receives no messages will have its test flag cleared.

The argument *time_limit* behaves as described above, in the description of `CM_send`.

These functions require $c + 3$ bits of temporary storage, where c is the value of `CM_cube_address_length`. The size of the gap must be at least this large when they are performed, and the contents of the gap may be arbitrarily altered.

CM_get(destination, cube_address, source, length)
```
CM_memaddr_t destination, cube_address, source;
unsigned length;
```

For every selected processor p_d, a message *length* bits long is sent to p_d from the processor p_S whose absolute cube address is stored at location *cube_address* in the memory of processor p_d. The message is taken from the field starting at location *source* within processor p_S and is stored into the field at location *destination* within processor p_d.

Note that more than one selected processor may request data from the same source processor p_S, in which case the same data is sent to each of the requesting processors.

The *length* must be not greater than the value of `CM_maximum_message_length`.

This function is conditional, but whether a message is sent depends only on the context flag of the requesting processor p_d; the message is transmitted regardless of the context flag of the source processor p_S.

Collisions, that is, two processors writing to the same processor, cannot occur with this function. The test flag is not affected.

This function requires $2c + 3$ bits of temporary storage, where c is the value of `CM_cube_address_length`. The size of the gap must be at least this large when it is performed, and the contents of the gap may be arbitrarily altered.

CM_fetch(destination, memory_address, length)

```
CM_memaddr_t destination, memory_address;
unsigned length;
```

For every selected processor p_d, an absolute virtual memory address stored in p_d at bit address *memory_address* specifies a source virtual processor p_S and a bit address b_S within that processor. A message *length* bits long is sent to p_d from the processor p_S. The message is taken from the field starting at location b_S within processor p_S and is stored into the field at location *destination* within processor p_d.

Note that more than one selected processor may request data from the same source processor p_S, possibly from the same field and possibly from different fields.

The *length* must be not greater than the value of `CM_maximum_message_length`.

This function is conditional, but whether a message is sent depends only on the context flag of the requesting processor p_d; the message is transmitted regardless of the context flag of the source processor p_S.

Collisions cannot occur with this function. The test flag is not affected.

This function requires $2c + v + 3$ bits of temporary storage, where c is the value of `CM_cube_address_length` and v is the value of `CM_virtual_memory_address_length`. The size of the gap must be at least this large when it is performed, and the contents of the gap may be arbitrarily altered.

Communication through the NEWS Grid

The communications functions in this section transmit data through the two-dimensional Connection Machine NEWS grid. They are considerably more efficient, when applicable, than using the general router mechanism.

CM_get_from_north(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

CM_get_from_east(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

CM_get_from_west(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

CM_get_from_south(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

Into the *destination* field of a given selected virtual processor is stored a copy of the *source* field of the virtual processor to its north, south, east, or west, respectively, regardless of whether the source processor is selected.

These functions may be used to operate on the flags.

The *length* must be not greater than the value of `CM_maximum_integer_length`.

These functions each require *length* bits of temporary storage. The size of the gap must be at least this large when they are performed, and the contents of the gap may be arbitrarily altered.

CM_get_from_north_always(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

CM_get_from_east_always(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

CM_get_from_west_always(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

CM_get_from_south_always(destination, source, length)
```
CM_memaddr_t destination, source;
unsigned length;
```

Into the *destination* field of a given virtual processor is stored a copy of the *source* field of the virtual processor to its north, south, east, or west, respectively, regardless of whether the source or destination processor is selected.

These functions may be used to operate on the flags.

The *length* must be not greater than the value of `CM_maximum_integer_length`.

Addresses and Address Transformations

CM_my_cube_address(destination)
```
CM_memaddr_t destination;
```

For each selected processor, the absolute virtual cube address of that processor is stored into the *destination*. The number of bits stored depends on the number of virtual processors in use; this number is made available to the user in the global variable CM_cube_address_length, which is initialized by the function CM_init.

CM_my_x_address(destination)
```
CM_memaddr_t destination;
```

CM_my_y_address(destination)
```
CM_memaddr_t destination;
```

For each selected processor, the virtual NEWS x-address or y-address of that processor is stored into the *destination*. The number of bits stored depends on the number of virtual processors in use; the numbers are made available to the user in the global variables CM_x_news_address_length, and CM_y_news_address_length, which are initialized by the function CM_init.

CM_x_from_cube(destination, source)
```
CM_memaddr_t destination, source;
```

CM_y_from_cube(destination, source)
```
CM_memaddr_t destination, source;
```

For each selected processor, a virtual cube address is taken from the *source* field and the corresponding virtual x-address or y-address is extracted and stored into the *destination* field. The number of bits read and stored depends on the number of virtual processors in use; the numbers are made available to the user in the global variables CM_cube_address_length, CM_x_news_address_length, and CM_y_news_address_length, which are initialized by the function CM_init.

CM_front_end_x_from_cube(cube_address)
```
CM_memaddr_t cube_address;
```

CM_front_end_y_from_cube(cube_address)
```
CM_memaddr_t cube_address;
```

Each of these functions is performed entirely within the front-end computer. The argument is taken to be a virtual cube address. The corresponding virtual x address or y address is extracted and returned.

CM_cube_from_x_y(destination, x_source, y_source)
```
CM_memaddr_t destination, x_source, y_source;
```

For each selected processor, virtual x and y addresses are taken from the *x_source* and *y_source* fields and the corresponding virtual cube address is constructed and stored into the *destination* field. The number of bits read and stored depends on the number of virtual processors in use; the numbers are made available to the user in the global variables CM_cube_address_length, CM_x_news_address_length, and CM_y_news_address_length, which are initialized by the function CM_init.

```
CM_cubeaddr_t
CM_front_end_cube_from_x_y(x_address, y_address)
unsigned x_address, y_address;
```

This function is performed entirely within the front-end computer. The arguments, non-negative integers, are taken to be virtual x and y addresses within the virtual NEWS grid. The corresponding virtual cube address is constructed and returned.

```
CM_enumerate_for_rendezvous(destination)
CM_memaddr_t destination;
```

For each selected processor, a different virtual cube address is stored in the *destination* field. These cube addresses are heuristically chosen, in a consistent manner, to try to optimize router performance in the event that a message is sent to each of the chosen addresses simultaneously. This function is almost exactly like `CM_enumerate` (and indeed could correctly be implemented simply by using `CM_enumerate`), except it does not guarantee to order the processors according to their virtual cube addresses.

Suppose two different sets of processors A and B are enumerated for rendezvous. Let E_A be the set of cube addresses assigned to the processors in A, and let E_B be the set of cube addresses assigned to the processors in B. Without loss of generality assume that $|A| \leq |B|$. Then it is guaranteed that $E_A \subseteq E_B$. The processors at the assigned cube addresses can then be used as intermediaries for parallel communication.

```
CM_processor_cons(destination, source)
CM_memaddr_t destination, source;
```

For each selected processor, the virtual cube address of a distinct *free* processor is (perhaps) stored in the *destination* field.

A processor is considered to be *free* if its one-bit *source* field contains a 1, regardless of whether or not it is selected. As much as possible, free processors are paired with selected processors, and for each pair three actions occur:

1. The selected processor receives the address of the free processor.
2. The test flag of the selected processor is set.
3. The *source* bit of the free processor is set to 0.

Any selected processor that is not paired with a free processor has its test flag cleared. The test flag of an unselected processor is not affected.

The *source* bit of any free processor that is not paired with a selected processor is not affected.

It is possible for a given processor to be both selected and free, in which case it might or might not be paired with itself. If a processor is both selected and free, then it will receive the address of some other processor—if enough are available—because it is selected, and some processor—perhaps itself, perhaps not—may receive its address, because it is free. It is expected that for many applications selected processors will not be free; however, it is up to the user to determine whether allowing a processor to have both roles simultaneously is meaningful within his or her application.

This function requires $8c + 13$ bits of temporary storage, where c is the value of `CM_cube_address_length`. The size of the gap must be at least this large when it is performed, and the contents of the gap may be arbitrarily altered.

Chapter 10
Memory Data Transfers

Transfer of Single Items

The functions described in this section transfer a single data item to or from a single field in a single specified processor.

```
long
CM_read_from_processor(address, source, length)
CM_cubeaddr_t address;
CM_memaddr_t source;
unsigned length;

CM_read_string_from_processor(dest, address, source, length)
char *dest;
CM_cubeaddr_t address;
CM_memaddr_t source;
unsigned length;

unsigned long
CM_u_read_from_processor(address, source, length)
CM_cubeaddr_t address;
CM_memaddr_t source;
unsigned length;

double
CM_f_read_from_processor(address, source, signif_len, expt_len)
CM_cubeaddr_t address;
CM_memaddr_t source;
unsigned signif_len, expt_len;
```

A field of bits, starting at virtual memory address *source* and continuing for *length* bits, is read from the memory of the virtual processor whose virtual cube address is *address*. This operation is performed unconditionally, without regard for the context flag of the processor.

For `CM_read_from_processor`, this field of bits is interpreted as a signed integer, and is returned as a long integer.

For `CM_u_read_from_processor`, this field of bits is interpreted as an unsigned integer, and is is returned as an unsigned long integer. This function may be used to read the flags, provided that *length* is equal to one.

For `CM_f_read_from_processor`, this field of bits is interpreted as a floating-point number, and is returned as a floating-point number. Overflow may occur in reducing the number read to fit the format of the front-end computer. In this case the global variable `CM_conversion_overflow` is set.

CM_write_to_processor(address, destination, value, length)
```
CM_memaddr_t address, destination;
int value;
unsigned length;
```

CM_write_string_to_processor(destination, address, source, length)
```
char *dest;
CM_cubeaddr_t address;
CM_memaddr_t source;
unsigned length;
```

CM_u_write_to_processor(address, destination, value, length)
```
CM_memaddr_t address, destination;
unsigned value, length;
```

CM_f_write_to_processor(address, destination, value, signif_len, expt_len)
```
CM_memaddr_t address, destination;
float value;
unsigned signif_len, expt_len;
```

The specified *value* is stored into a field of bits, starting at virtual memory address *destination* and continuing for *length* bits, within the memory of the virtual processor whose virtual cube address is *address*. This operation is performed unconditionally, without regard for the context flag of the processor.

For `CM_write_to_processor`, the value should be an integer; the low-order bits of the integer are stored.

The function `CM_u_write_to_processor` behaves identically to `CM_write_to_processor`, and is provided primarily for reasons of symmetry. It may be used to write the flags, provided that *length* is equal to one.

For `CM_f_write_to_processor`, *value* should be a floating-point number. Overflow may occur in reducing the number read to fit the format of the specified field, but such overflow is not detected.

Transfer of Arrays

The array transfer functions form a symmetrical group arranged according to three categories:

Data type. This refers to the type of item transferred.

- —Signed integers
- —Unsigned integers
- —Floating-point numbers

Direction.

- —Functions with `read` in their names transfer data from virtual processors in the Connection Machine system to an array in the front-end computer.
- —Functions with `write` in their names transfer data from an array in the front end to virtual processors in the Connection Machine system.

Order. Every array transfer transfers one item to or from each virtual processor within a set of virtual processors. This set may be designated in either of two ways:

- —A linear range of virtual cube addresses
- —A subrectangle of the virtual NEWS grid

```
CM_read_array_by_cube_addresses(array, array_element_type, array_offset,
        cube_address_start, cube_address_end,
        source, length)
char *array;
unsigned array_element_type, array_offset, length;
CM_cubeaddr_t cube_address_start, cube_address_end;
CM_memaddr_t source;

CM_write_array_by_cube_addresses(array, array_element_type, array_offset,
        cube_address_start, cube_address_end,
        destination, length)
char *array;
unsigned array_element_type, array_offset, length;
CM_cubeaddr_t cube_address_start, cube_address_end
CM_memaddr_t destination;

CM_read_array_by_news_addresses(array, array_row_size, array_element_type,
        array_x_offset, array_y_offset, x_start, y_start, x_end, y_end,
        source, length)
char *array;
unsigned array_row_size, array_element_type,
        array_x_offset, array_y_offset, x_start, y_start, x_end, y_end,
        length;
CM_memaddr_t source;

CM_write_array_by_news_addresses(array, array_row_size, array_element_type,
        array_x_offset, array_y_offset, x_start, y_start, x_end, y_end,
        destination, length)
char *array;
unsigned array_row_size, array_element_type,
        array_x_offset, array_y_offset, x_start, y_start, x_end, y_end,
        length;
CM_memaddr_t destination;
```

Signed integer values are transferred between the specified *array* and Connection Machine processors.

The *array* argument is a C pointer; the array to which it points should be one-dimensional for cube-oriented transfers, and two-dimensional for NEWS-oriented transfers. The argument *array_row_size* specifies the last dimension of the two-dimensional array. The *array_element_type* is a value indicating the base type—the bit length—of the array elements. Its value must be one of `CM_bit_array`, `CM_byte_array`, `CM_short_array`, or `CM_long_array`, for 1-, 8-, 16-, and 32-bit array elements, respectively.

For cube-oriented transfers of one-dimensional arrays, an index *j* ranges from 0 to *cube_address_end* − *cube_address_start* − 1, inclusive. The data from virtual processor *cube_address_start* + j is transferred to or from array element *array_offset* + j. Note that *cube_address_start* is an inclusive address (the address of the first processor involved in the transfer), but *cube_address_end* is an exclusive address (one more than the address of the last processor involved in the transfer). The total number of values transferred is therefore *cube_address_end* − *cube_address_start*.

For NEWS-oriented transfers of two-dimensional arrays, indices *j* and *k* range respectively from 0 to *x_end* − *x_start* − 1, inclusive, and from 0 to *y_end* − *y_start* − 1, inclusive. The data from the virtual processor whose NEWS coordinates are (*x_start* + j, *y_start* + k) is transferred to or from the array element whose indices are (*array_x_offset* + j, *array_y_offset* +

k). Note that *x_start* and *y_start* are inclusive addresses, but *x_end* and *y_end* are exclusive addresses. The total number of values transferred is therefore (*x_end* − *x_start*) × (*y_end* − *y_start*).

Such a transfer copies a subrectangle of the virtual NEWS grid to or from a subrectangle of an array in the front end. This can be particularly useful for copying to or from Connection Machine memory an image to be displayed on a bit-mapped raster-scan graphics terminal.

The *source* or *destination* argument specifies the memory address within each processor of the field to be transferred, considered as a signed integer. The *length* argument specifies the length of the field.

For `write` operations, the overflow flag is set in any processor for which the transferred integer cannot be represented in the specified field, and is cleared in any processor for which the transferred integer can be represented. The overflow flag is unaffected in any processor not involved in the transfer.

Overflow can occur for `read` operations; no warning is given.

These operations are unconditional, performed without regard to the context flag.

```
CM_u_read_array_by_cube_addresses(array, array_element_type, array_offset,
        cube_address_start, cube_address_end, source, length)
char *array;
unsigned array_element_type, array_offset, length;
CM_cubeaddr_t cube_address_start, cube_address_end;
CM_memaddr_t source;

CM_u_write_array_by_cube_addresses(array, array_element_type, array_offset,
        cube_address_start, cube_address_end, destination, length)
char *array;
unsigned array_element_type, array_offset, length;
CM_cubeaddr_t cube_address_start, cube_address_end
CM_memaddr_t destination;

CM_u_read_array_by_news_addresses(array, array_row_size, array_element_type,
        array_x_offset, array_y_offset, x_start, y_start, x_end, y_end, source, length)
char *array;
unsigned array_row_size, array_element_type,
        array_x_offset, array_y_offset, x_start, y_start, x_end, y_end,
        length;
CM_memaddr_t source;

CM_u_write_array_by_news_addresses(array, array_row_size, array_element_type,
        array_x_offset, array_y_offset, x_start, y_start, x_end, y_end,
        destination, length)
char *array;
unsigned array_row_size, array_element_type,
        array_x_offset, array_y_offset, x_start, y_start, x_end, y_end,
        length;
CM_memaddr_t destination;
```

Unsigned integer values are transferred between the specified *array* and Connection Machine processors. In all other respects, these functions and their arguments are exactly as described above for signed integers.

For `write` operations, the overflow flag is set in any processor for which the transferred integer cannot be represented in the specified field, and is cleared in any processor for which

the transferred integer can be represented. The overflow flag is unaffected in any processor not involved in the transfer.

Overflow can occur for read operations; no warning is given.

These operations are unconditional, performed without regard to the context flag.

```
CM_f_read_array_by_cube_addresses(array, array_element_type, array_offset,
        cube_address_start, cube_address_end,
        source, signif_len, expt_len)
char *array;
unsigned array_element_type, array_offset, signif_len, expt_len;
CM_cubeaddr_t cube_address_start, cube_address_end;
CM_memaddr_t source;

CM_f_write_array_by_cube_addresses(array, array_element_type, array_offset,
        cube_address_start, cube_address_end,
        destination, signif_len, expt_len)
char *array;
unsigned array_element_type, array_offset, signif_len, expt_len;
CM_cubeaddr_t cube_address_start, cube_address_end
CM_memaddr_t destination;

CM_f_read_array_by_news_addresses(array, array_row_size, array_element_type,
        array_x_offset, array_y_offset, x_start, y_start, x_end, y_end,
        source, signif_len, expt_len)
char *array;
unsigned array_row_size, array_element_type,
        array_x_offset, array_y_offset, x_start, y_start, x_end, y_end,
        signif_len, expt_len;
CM_memaddr_t source;

CM_f_write_array_by_news_addresses(array, array_row_size, array_element_type,
        array_x_offset, array_y_offset, x_start, y_start, x_end, y_end,
        destination, signif_len, expt_len)
char *array;
unsigned array_row_size, array_element_type,
        array_x_offset, array_y_offset, x_start, y_start, x_end, y_end,
        signif_len, expt_len;
CM_memaddr_t destination;
```

Floating-point values are transferred between the specified *array* and Connection Machine processors. In all respects, except for the arguments *signif_len*, *expt_len*, and *array_element_type* described below, these functions and their arguments are exactly as described above for signed integers.

Instead of a single *length* argument, the two arguments *signif_len* and *expt_len* specify the floating-point format, as usual. The *array_element_type* is a value indicating the base type of the source or destination array elements. Its value must be either `CM_float_single_array` for a base type of `single`, or `CM_float_double_array` for a base type of `double`.

For `write` operations, the overflow flag is set in any processor for which the transferred floating-point value cannot be represented in the specified field, and is cleared in any processor for which the transferred value can be represented. The overflow flag is unaffected in any processor not involved in the transfer.

Overflow *can* occur for `read` operations; no indication of this condition is given.

These functions are unconditional, performed without regard to the context flag.

Index

G

I

L

M

N

O

P

R

S

T

U

V

W

X

Y

Introduction to Data Level Parallelism

Thinking Machines Technical Report 86.14

Introduction to Data Level Parallelism

With Programming Examples
for the Connection Machine® System

April 1986

Contents

List of Figures

Chapter 1

Data Level Parallelism

1.1 Parallelism in the World Around Us

Whenever many things happen at once, parallelism is at work. It is at work for one of two reasons: either because someone is in a hurry or because it is the natural course of events. If, for example, many people are working at once to compose a song, it is because someone is in a hurry. Music is a naturally sequential process. Physical phenomena, on the other hand, are almost always parallel. The wind in a wind tunnel does not blow over one square centimeter of an automobile body at a time. It blows across the whole frame at once, showing the engineers how the flow in one section interacts with the flow in another. If we simulate the wind in parallel, the results come faster as a natural consequence. The parallelism is being utilized, but it is not being artificially imposed. Other examples of fundamentally parallel phenomena include vision processing, information retrieval, and many types of mathematical operations.

1.2 Parallelism in Computer Systems

The same two motivations, doing things in a hurry and doing things more naturally, also motivate computer architects. Until recently, those architects who are focused on greater speed have obtained it from faster circuitry. Making the electronics twice as fast, or the memory twice as big, has traditionally been a cost-effective way to double the performance of a single-processor computer system. But now these gains have become much harder to achieve. Limits to circuit speed have been reached. So designers who are solely focused on speed are now seeking to inject parallelism into their designs. If two computers of traditional architecture can operate in parallel, the overall speed of the system can double.

There is, however, another starting point for the design process. Computer architects

can go back to the problems themselves and understand the parallelism that has been there all along. Having understood it, they can build a system that exploits it directly. The first benefit of this approach is simplicity. A computer that fits the problems it solves is easier to use and program than a computer that doesn't. And it is also faster. Systems that couple to the inherent structure of a problem mine a deeper vein of parallelism. For this reason, they can dramatically outperform systems whose superficial performance specifications seem superior. When parallelism is imposed on a problem, a speed-up of ten is considered good. When inherent parallelism is exploited, speed-ups of 1000 are commonplace.

Some applications benefit much more than others. While certain problems do not have a large amount of parallelism, there is a large and growing body of important problems that do. For these applications the method of designing the computer around the inherent parallelism of the problem is proving to be outstandingly valuable. This approach is called "data level parallelism." The remaining sections of this report describe data level parallelism and its application to three very different computing problems. The implementation examples use the Connection Machine system, the first data level parallel computer available on the commercial market. (See reference [8] for further discussion of the Connection Machine system)

1.3 Two Styles of Computer Parallelism

All computer programs consist of a sequence of instructions (the control sequence) and a sequence of data elements. Large programs have tens of thousands of instructions operating on tens of thousands, or even millions of data elements. Parallelism exists in both places. Many of the instructions in the control sequence are independent; they may in fact be executed in parallel by multiple processors. This approach is called "control level parallelism." On the other hand, large numbers of the data elements are also independent; operations on these data elements may be carried out in parallel by multiple processors. This approach, as mentioned in the previous section, is called "data level parallelism." Each approach has its strengths and limitations. In particular, data level parallelism works best on problems with large amounts of data. Small data structures generally do not have enough inherent parallelism at the data level. When the ratio of program to data is high, it is often more efficient to use control level parallelism. But control level parallelism requires the user to break up the program and then maintain control and synchronization of the pieces.

1.4 The Connection Machine Data Level Parallel Computer

The Connection Machine computer from Thinking Machines Corporation is the first system to implement data level parallelism in a general purpose way. Since the computer is designed

around the structure of real world problems, the best way to understand the Connection Machine architecture is to follow its use in solving an actual problem. A VLSI simulation example will be used for that purpose. In VLSI simulation, the computer is used to verify a circuit design before it is released to be manufactured. The Connection Machine system provides a very direct way to perform this simulation. Each transistor in the circuit is simulated by an individual processor in the system. The chapters which follow explain three more examples in much greater detail.

1.4.1 Program Execution

Data level parallelism uses a single control sequence, or program, and executes it one step at a time, just as it is done on a traditional computer. The Connection Machine system utilizes a standard architecture front end computer for this purpose. All programs are stored on the front end machine. Its operating system supports program development, networking, and low speed I/O. The front end computer has access to all the memory in the system, albeit one data element at a time because it is a serial computer.

All Connection Machine program execution is controlled by the front end system. A Connection Machine program has two kinds of instructions in it: those that operate on one data element and those that operate on a whole data set at once. Any single-data-element instructions are executed directly by the front end; that is what it is good at. The important instructions, those that operate on the whole data set at once, are passed to the Connection Machine hardware for execution.

In the VLSI simulation example, the important instructions are the ones which tell each processor to step through its individual transistor simulation process. Each processor executes the same sequence of instructions, but applies them to its own data, the data that describes the voltage, current, conductance, and charge of its transistor at that time step of the simulation.

1.4.2 The Connection Machine Processors

In order to operate on the whole data set at once, the Connection Machine system has a distinct processor for each data element. The system implements a network of 65,536 individual computers, each with its own 4096 bits of memory. The data that describe the problem are stored in the individual processors' memories. During program execution, whenever the front end encounters an instruction which applies to all the data at once, it passes the instruction across an interface to the Connection Machine hardware. The instruction is broadcast to all 65,536 processors, which execute it in parallel.

Applications problems need not have exactly 65,536 data items. If there are fewer, the system temporarily switches off the processors that are not needed. If there are more problem elements, the Connection Machine hardware operates in virtual processor mode.

Each physical processor simulates multiple processors, each with a smaller memory. Virtual processing is a standard, and transparent, feature of the system. A Connection Machine system can easily support up to a million virtual processors. In general, a problem should have between ten thousand and a million data elements to be appropriate for the Connection Machine system.

1.4.3 Connection Machine I/O

Since the front end system has access to all Connection Machine memory, it can load data into that memory and read it back out again. For small amounts of data, this is a practical approach, but for large amounts it is too slow. A separate 500-megabit-per-second I/O bus is used instead. This bus is used for disk swapping, image transfer, and other operations which exceed the capacity of the front end.

1.5 Communications: The Key to Data Level Parallelism

Large numbers of individual processors are necessary for data level parallelism, but by themselves they are not enough. After all, there is more to a VLSI circuit than individual transistors. A circuit is made up of transistors connected by wires. Similarly, there is more to a Connection Machine system than just processors. A Connection Machine system is made up of processors interconnected by a massive inter-connection system called the router.

The router allows any processor to establish a link to any other processor. In the case of the VLSI simulation example, the links between processors exactly match the wiring pattern between the transistors. Each processor computes the state of an individual transistor and communicates that state to the other processors (transistors) it is connected to. All Connection Machine processors may send and receive messages simultaneously. The router has an overall capacity of three billion bits per second.

It is part of the reality of the world we live in that many things happen at once, in parallel. It is part of the beauty of the world we live in that these many things connect and interact in a variety of patterns. Looking at the whole problem at once requires a computer that combines the ability to operate in parallel with the ability to interconnect.

Since the structure of each problem is different, the interconnection pattern of the computer must be flexible. All linkages between Connection Machine processors are established in software. Therefore, the system can configure its processors in a rectangular grid for one problem and then into a semantic network for the next. Rings, trees, and butterflies are other commonly used topologies. The chapter on hardware describes router operation in greater detail.

1.6 Connection Machine Application Examples

Each of chapters 2, 3, and 4 describes a Connection Machine example in detail. First the algorithm is described, and then the actual program that implements this algorithm is presented and discussed. It is not necessary to study the program to appreciate the simplicity of the overall approach. Many readers will want to skip over these details. The third example, contour mapping, is quite sophisticated. Hence the program for this example is more complex than the two that precede it.

The initial Connection Machine languages are C* and *Lisp. C* is an extension of C and is appropriate for a wide range of general purpose applications. *Lisp is an extension of Lisp. Lisp, while less well known than C, is also an appropriate language for a wide variety of applications. Its primary use, however, has been in the field of artificial intelligence. Chapters 5 and 6 provide an introduction to these languages.

Chapter 2

Document Retrieval

There is too much to read. The written material for almost every discipline grows much faster than any one person can read it. Computers have not provided much relief to date. Now data level parallelism provides the computing power to implement significantly better solutions to the document retrieval problem. These solutions are more natural, so they require less user training. And they are much more accurate, so they give the user much greater confidence in the results.

2.1 Accessing Computer Data Bases

There are a number of systems today that provide on-line access to text information, but they perform poorly because they rely on a "keyword" mechanism for finding documents. The premise of a keyword system is that the relevance of a whole document can be determined by the presence or absence of a few individual words. Users enter one or more "keywords" or labels that they feel capture the sense of the information needed. All documents which either contain these words or have been indexed under these words are retrieved. Those that do not are ignored. Even with refinements, such as "Find all occurrences of 'New England Patriots' within ten words of 'Superbowl'," a keyword search generally tends to either find too many documents for the user to cope with, or too few for the user to find useful. It is a guessing game, with the user trying to imagine the most fruitful search terms.

Not all relevant documents contain the one particular word that the user chose, because writers use language differently. A search for documents containing the word "chips" may find five relevant documents, but miss ten others that were indexed under "integrated circuits" or "VLSI." Since the search yields only one third of the relevant documents, it would be considered to have a *recall* of 33%. Worse yet, the five relevant documents might be returned mixed into twenty other documents describing cookies or paint or other subjects

where the word "chips" appears. Such a search would be considered to have a *precision* of 20%. Recent published testing has shown that recall results of as little as 20% are common with keyword based systems [1].

In short, keyword-based systems are very good at finding one or two relevant documents quickly. What they are poor at is producing a refined result with high recall and high precision. The Connection Machine document retrieval system provides a very powerful way for doing complete searches. It starts out using a keyword approach, but once the first relevant document is found, the whole approach changes. The user proceeds by simply pointing to one or more relevant documents and saying, in effect, "Find me all the documents in the database that are on the same subjects as this one." A document that has been identified as relevant by the user is referred to here as a "good document."

2.2 Algorithms for Document Retrieval

Data level parallelism makes massive document comparisons simple. The basic idea is this: a database of documents is stored in the Connection Machine system, one or more documents per processor. Once the first good document is found, it is used to form a search pattern. The search pattern contains all the content words of the document. The host machine broadcasts the words in the pattern to all the processors at once. Each processor checks to see if its document has the word. If it does, it increases the score for its document. When the entire pattern has been broadcast, the document that most closely matches the pattern will have the highest score, and can be presented first to the user.

The algorithm is simple to program because it takes advantage of innate characteristics of documents rather than programming tricks and second guessing. Every document is, in effect, a thesaurus of its subject matter. A high percentage of the synonyms of each topic appear because writers work to avoid repetition. In addition, variants of each word (such as plural, singular, and possessive forms), and semantically related terms also appear among the words in a particular article. Clearly not every synonym, variant, and related term will occur in a single article, but many terms will. Each reinforces the connection between the search pattern and the document. Spurious documents, on the other hand, will not be reinforced. The word "chip" will appear in an article about cookies, but "VLSI" and "integrated circuit" simply will not. In the overall scoring, truly useful documents are reliably separated from random matches. (See figures 2.1 and 2.2.)

2.3 Database Loading on the Connection Machine System

A document database may be constructed from sources of text such as wire services, electronic mail, and other electronic databases. For this description it is important to draw a

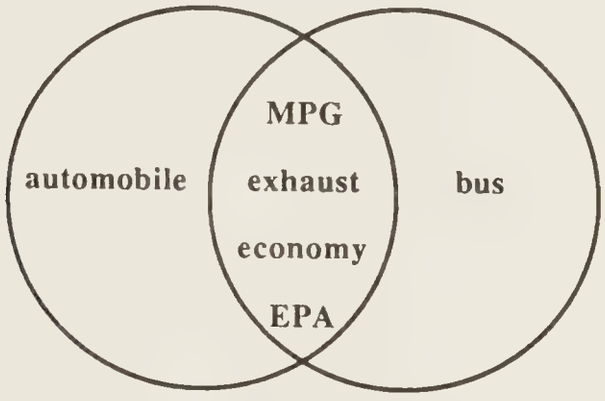


Figure 2.1: *Documents on the same subject have a high overlap of vocabulary.*

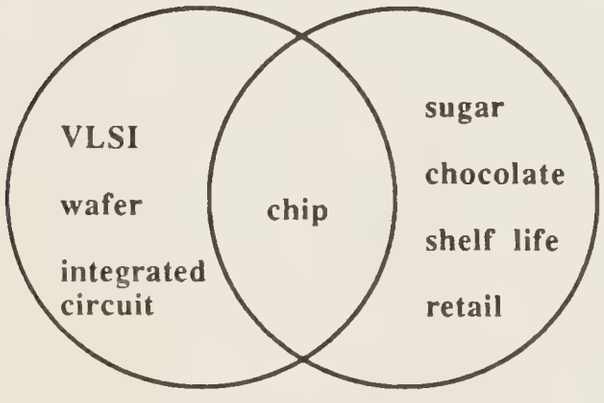


Figure 2.2: *Documents on different subjects have low overlap of vocabulary.*

distinction between *source documents* and *content kernels*. A *source document* contains the full actual text of a particular article, book, letter, or report, and is stored on the front-end's disk. A *content kernel* is a compressed form of the source document that encodes just the important words and phrases. It omits the commonplace words. Content kernels are stored in the memory of Connection Machine system.

The content kernel is produced automatically from the source document. First, the source document is processed by a Thinking Machines document indexer program that marks the most significant terms in the text. Next these terms are encoded into a bit-vector data structure, using a method called "surrogate coding." Surrogate coding, which is sometimes referred to as a "hash coding" method, allows the content kernel to be stored more compactly. It also speeds up the search process. In surrogate coding, each term in the content kernel is mapped into ten different bits in a 1024-bit vector. The ten selected bits in the vector are set to one to indicate the presence of the word in the document. In a content kernel of 30 terms, the process of surrogate coding ends up marking about a third of the bits as ones.

The source document in its original form is available for retrieval and presentation to the user when needed. The location of the original document on the system disk is stored with the content kernel.

Each segment of the content kernel is made up of the following fields:

`*score*` is used by the document lookup program to accumulate the ranking of each content kernel in the database according to how closely the content kernel matches the user's search pattern. Each time a match is found, `*score*` is updated.

`*document-id*` contains a reference to the original source document that this content kernel was derived from. When a content kernel is selected from the database lookup, the user is shown the source document referred to by this index.

`*kernel*` is a table of the surrogate-coded bit-vector encoding.

The necessary declarations for these fields are as follows. (In this chapter only, all of the code is presented twice, first in the *Lisp language and then in the C* language, to make it easy to compare the two languages. Because the characters * and ? may not appear in C* identifiers, such *Lisp names as `*score*` and `word-appears?` are rendered in C* simply as `score` and `word_appears`.)

```
;;; Declarations for the *Lisp version.

(defconstant table-size 1024)
(defconstant hash-size 10)
```

```
(*defvar *score*)
(*defvar *document-id*)
(*defvar *kernel*)


/* Declarations for the C* version. */

#define TABLE_SIZE 1024
#define HASH_SIZE 10

poly unsigned score, document_id;

poly bit kernel[TABLE_SIZE];
```

2.4 Document Lookup on the Connection Machine System

During the first stage of document lookup, the user lists a set of terms to be used to search the database, and receives back an ordered list of documents that contain all or some of those terms. The user then points to a document which is relevant, and from this document an overall *search pattern* of content-bearing words is assembled. The search pattern is simply a list of these words, with weights assigned to each word. The weight assigned to a word is inversely proportional to its frequency in the database (for example, "platinum" appears in the database less frequently than "gold," and therefore has a higher weight associated with it). This weighting mechanism ensures that uncommon words have more of an influence than common words over which content kernels get selected during the document lookup process.

Next, the search pattern is broadcast to all processors in the Connection Machine system. The same mechanism that is used to code each word in the content kernel as a series of bits is applied to the words in the search pattern. For each word in the search pattern a set of ten bit indices is broadcast. All content kernels that have these same ten bits set will have the weight of that word added into their `*score*` field. (It is possible that all ten bits for a word might happen to be set on account of other words even though that word doesn't really appear in the source document. Such an accident will result in a "false hit" on that word. However, for two reasons, this will not seriously affect the results of the lookup. First, the probability of a false hit is small: $(\frac{1}{3})^{10}$, or less than one in 50,000. Second, a false hit will be only one of many terms contributing to the score, and so will have only a small effect even when it does occur.)

The following code is used to broadcast one search pattern word to all the processors

in the system, which check their content kernels and add the value of weight into their *score* if it contains the word. The word is represented by a list of ten bit locations (bit-locs).

```
;;; *Lisp code for testing the presence of a single word.

(*defun increment-score-if-word-appears (bit-locs word-weight)
  (*let ((word-appears? t!!))
    (dolist (bit bit-locs)
      (*set word-appears?
            (and!! word-appears?
                   (not!! (zerop!! (load-byte!! *kernel* (!! bit) (!! 1)))))))
      (*if word-appears?
           (*set *score* (+!! *score* (!! word-weight))))))


/* C* code for testing the presence of a single word. */

poly void increment_score_if_all_bits_set
        (mono unsigned word_bit_position[HASH_SIZE], mono int weight) {
    mono j;
    poly bit word_appears = 1;
    for (j = 0; j < HASH_SIZE; j++)
        word_appears &= kernel[word_bit_position[j]];
    if (word_appears)
        score += weight;
}
```

The main search program simply calls this routine once for each keyword in the keyword list.

2.5 Retrieving the Highest Scoring Documents

The code that follows is used to retrieve the *document-id* for each of the highest scoring content kernels in the database. The program returns a list of *document-id*s for the content kernels with the highest scores. The program first retrieves the *document-id* for the highest score, then the next highest score, etc., until a list of length document-count is retrieved. The already-retrieved? flag is set once a processor has had its *document-id* retrieved so it will not be retrieved again.

```
;;; *Lisp code for retrieving documents in order, highest score first.

(*defun retrieve-best-documents
  (let ((top-documents-list nil))
    (*let ((already-retrieved? nil))
      (dotimes (i document-count)
        (*when (not!! already-retrieved?)
          (*when (=!! *score* (*max *score*))
            (*let ((next-highest-document (*min (self-address!!))))
              (setq top-documents-list
                (append top-documents-list
                  (list (pref *document-id* next-highest-document))))
                (setf (pref already-retrieved? next-highest-document) t))))))
    top-documents-list))


/* C* code for retrieving documents in order, highest score first. */

poly void retrieve_best_documents
          (mono document_count, mono unsigned *document_id_array) {
    poly bit already_retrieved = 0;
    mono i;
    for (i = 0; i < document_count; i++) {
        if (!already_retrieved) {
            if (score == (><= score)) {
                processor *next_highest_document = (<>= this);
                document_id_array[i] = next_highest_document->document_id;
                next_highest_document->already_retrieved = 1;
            }
        }
    }
}
```

2.6 Timing and Performance

A production level version of the algorithms described above has been implemented and extensively tested on the Connection Machine system. Performance studies have been done on a database of 15,000 newswire articles, which constitute 40 megabytes of text. An

automatic indexing system, selects the content kernels for each document. The content kernels are about one third of the original size of the text. Surrogate coding compresses the data by another factor of about two. In the system currently in use, the kernels are encoded into as many 1024-bit vectors as are needed at 30 terms per vector. For a long document several vectors are used; additional code, not shown above, is needed to chain the vectors together and combine the results.

Using this encoding, the Connection Machine system is able to retrieve the 20 nearest documents to a 200-word search pattern from a data base of 160 MBytes in about 50 milliseconds. (160 MBytes is equivalent to an entire year of news from a typical newswire.) In this time the Connection Machine system performs approximately 200 million operations for an effective execution speed of 6,000 Mips.

2.7 Summary and Implications

The program is brief because the algorithm is simple. The Connection Machine system is able to match the user's needs directly. It is powerful enough to carry out the algorithm in a straightforward way. The user wants to say to the database "All documents on the same subject as this one, line up in order here." That is exactly the service that the Connection Machine system provides for the user. It broadcasts the contents of the selected document to tens of thousands of processors at once. Each processor decides in parallel how similar its documents are. Then the most similar ones are sorted and presented to the user.

Even larger databases can use the same technique with two enhancements. The first enhancement is the use of a very high-speed paging disk, which allows larger numbers of content kernels to be swapped into the system for searching. The second enhancement is the use of cluster analysis. When the system has many documents on the same subject, it need not store all their content kernels individually. It can store one for the whole cluster, then retrieve the full set of related documents when needed. A single document may, of course, participate in more than one cluster. As the total database size grows, the size of the average cluster grows with it, making this a particularly appropriate technique for large scale databases. The addition of paging and clustering extends the algorithm described above to the 10-gigabyte range and beyond.

Chapter 3

Fluid Dynamics

Fluid flow simulation is a key problem in many technological applications. From the flow of air over an airplane wing to mixing in a combustion chamber, the problem is to predict the performance of a design without building and testing a physical model.

Until recently, fluid flow models were based almost exclusively on partial differential equations, typically the Navier-Stokes equations or approximations to them. These equations are not generally solvable by normal analytical methods. Numerical approximation techniques, such as finite difference methods and finite element methods, have been developed to solve these partial differential equations. All of these methods involve large numbers of floating point operations which require great amounts of fast memory. In addition, obstructions to the flow must usually be mathematically simple shapes.

Recent physics research has suggested that it is possible to make intrinsically discrete models of fluids. The fluids are made up of idealized molecules that move according to very simple rules, much simpler than the Navier-Stokes equations. The models are examples of cellular automata and are particularly well-suited to simulation on the Connection Machine. Cellular automata are systems composed of many cells, each cell having a small number of possible states. The states of all cells are simultaneously updated at each "tick" of a clock according to a simple set of rules that are applied to each cell. This approach involves only simple logical operations and does not require floating point arithmetic. It allows for all obstructions regardless of their shape. In addition, mathematical methods can be used to show that the results of such simulations agree with the results that would be obtained from the Navier-Stokes equations.

3.1 The Method of Discrete Simulation

Discrete simulation is used to model fluid flow on the Connection Machine system. The technique involves six key elements: particles, cells, time steps, states, obstacles, and interaction rules. *Particles* correspond to molecules of a fluid. A particle has a speed and a direction which determine how it moves. A *time step* is a "tick" of a clock that synchronizes the movement of particles. During each time step, particles move one cell in the direction that they are heading. A *cell* is a specific place in the overall region that is being observed. The region is completely filled with cells. Particles can move into and out of each cell during each time step. A *state* is a value assigned to each cell that indicates the number of particles within the cell, and in which directions they are heading. An *obstacle* is a set of special cells that obstruct the natural movement of particles. The *interaction rules* determine the movement of each particle when it shares a cell with one or more other particles. This movement is carried out by updating the state of the cells to reflect the new positions of the particles within the region.

A discrete simulation typically uses fixed cells. The cells never move or change during the simulation. Particles are completely in one cell during a time step, and move completely into the next cell (determined by the interaction rules) during the next time step. During each time step, every cell gathers data about particles heading in its direction from each of its neighboring cells. Based on the interaction rules, each cell determines the direction of its newly acquired particles and updates its own state.

A simulation designer can choose the cell topology and the interaction rules. The cell topology determines how many sides a cell has, and therefore, the directions by which particles may enter and exit. The simulation designer also determines the number of cells in the region being observed, and the average number of particles in each cell. Cellular automata theory provides the background for the simulation designer's decisions. It suggests that a simple cell topology, a huge number of cells and particles, and simple, local interaction rules are the most likely to be successful.

3.2 A Discrete Simulation of Fluid Flow

Thinking Machines is currently simulating fluid flow using a two-dimensional region that is divided into 16,000,000 hexagonal cells. Each cell is assigned to its own Connection Machine processor (using the virtual processor mechanism). The hexagonal mesh is a simple topology that gives the randomness that is required on a microscopic level to get correct results on the macroscopic level.

One of the fundamental reasons for computer simulation of fluid flow is to observe the behavior of a fluid as it flows past an obstacle. In the discrete model, obstacles are groups of cells that particles can not travel through. When a particle approaches an obstacle cell,

it bounces off during the next time step. In order to observe the behavior of a fluid, tens of millions of microscopic particle interactions are simulated. Each individual particle's path through the cells and off of the obstacle cells appears almost random, just as in real fluids. However, when all of the particles' paths are considered, the overall behavior of the model is consistent with the way that real fluids behave. (See references [4,7,14] for further discussion of the use of cellular automata to model fluid flow.)

Individual particles can enter or exit through any of the six sides of each cell. A cell may contain a maximum of one particle heading in each of the six possible directions during a given time step (and so the total number of particles per cell per time step is anywhere from 0 to 6). A particle that has not collided with another particle during a time step will continue moving in the same direction during the next time step. (See figure 3.1.) When particles collide, a simple set of rules determines their new directions, conserving both momentum and the number of particles.

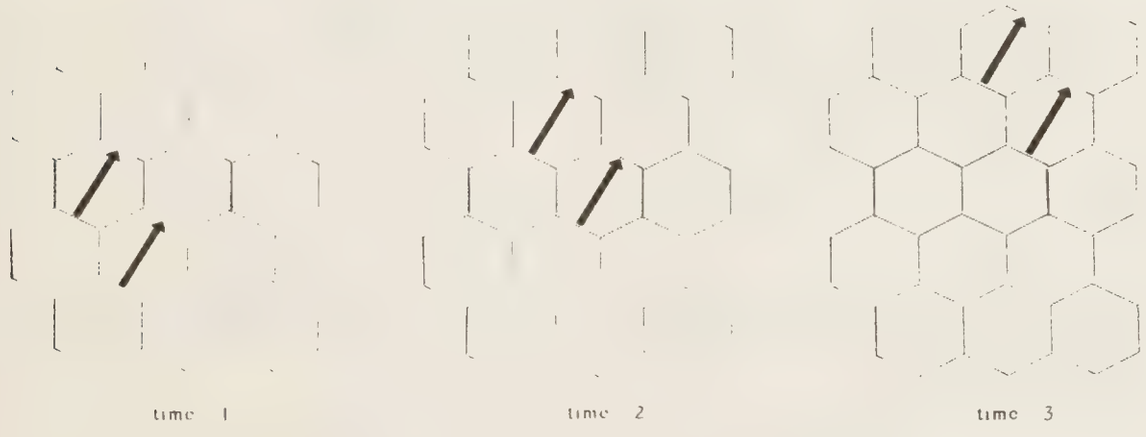


Figure 3.1: *Unless particles are obstructed by an obstacle, or collide into other particles, they continue in the same direction.*

At each time step, every cell updates its state by checking all of its adjoining cells, or neighbors, for particles that are heading in its direction. All cells then update their own states based on the information that they have gathered. In the model currently implemented, there are five situations that cause a particle to change directions: 2-way symmetric collisions, 3-way symmetric collisions, 3-way asymmetric collisions, 4-way symmetric collisions, and collisions with an obstacle cell. (See figure 3.2.)

Although the algorithm is implemented by modeling the individual movements and collisions of tens of millions of particles at each time step, the behavior of the fluid is observed by averaging the behavior of all of the particles in the entire region and by analyzing the

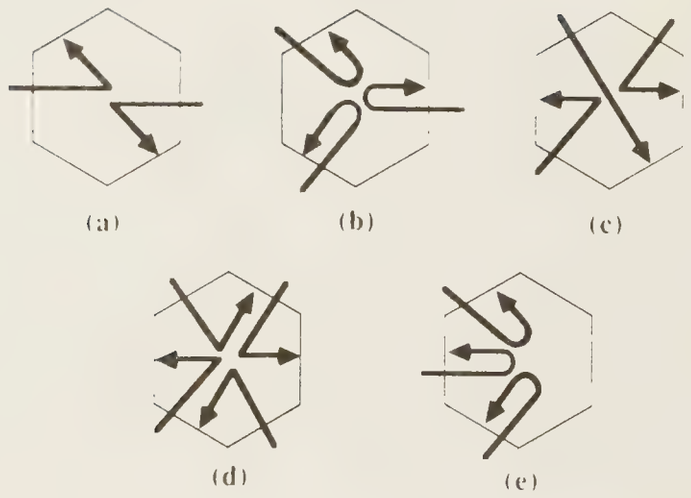


Figure 3.2: *Situations that cause particles to change directions.*
(a) *Two-way symmetric: two particles enter a cell from opposite sides. The particles exit through a different pair of opposite walls.*
(b) *Three-way symmetric: three particles enter a cell from non-adjacent sides. Each particle exits by the side through which it entered.*
(c) *Three-way asymmetric: three particles enter a cell, two of them from opposite sides. One particle passes through unobstructed; the other two particles behave as in a two-way symmetric.*
(d) *Four-way symmetric: four particles enter a cell, each particle's side is adjacent to only one other particle's side. Particles behave as in two two-way symmetric collisions (maximum of one particle exiting per side).*
(e) *Collisions with an obstacle cell: a particle always leaves an obstacle cell by the side through which it entered.*

results over many time steps. In a typical simulation, macroscopic results are gathered by averaging particles together in groups of 20,000. Although each individual particle has only one speed and six possible directions, the average of 20,000 particles provides the full range of possible velocities.

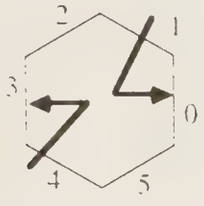
3.3 Implementation on the Connection Machine System

There are two available ways for the Connection Machine system to implement the connections among the hexagonal cells. It can use the full router, setting up six connections for each site, one for each adjacent hexagon. Or it can use its grid, which connects four

adjacent processors directly. The grid network was chosen for this implementation. It is very fast for small data transfers to nearby processors.

Of course, the grid cannot implement hexagonal connections directly. It connects to four adjacent processors, not six. Therefore, two of the six connections require two-step communication (i.e., up one and over one for the diagonal). The simulation program implements this two-step process. Each site can quickly learn the status of its six neighbors and can determine which ones contain particles that are moving in its direction.

Each cell has only 13 bits associated with it: six bits for incoming state (numbered 0–5), six bits for outgoing state (numbered 0–5), and one bit to indicate whether or not it is an obstacle. Each of the six incoming state and six outgoing state bits is dedicated to a particular direction. If a particle is entering or exiting through that direction, then the bit is set to 1, otherwise it is set to 0. (See figure 3.3.)



BITS	0	1	2	3	4	5
INCOMING	0	1	0	0	1	0
OUTGOING	1	0	0	1	0	0

Figure 3.3: *Hexagonal cells with six incoming bits for particle direction and six outgoing bits for particle direction*

```
/* A cell state is represented by a six-bit unsigned integer,
   which can also be regarded as an array of six individual bits. */

typedef union STATE {unsigned:6 Val; unsigned:1 Bit[6];} state;

/* Each processor in the domain "grid" will contain a cell state
   (the outgoing state), another state (the incoming state) used
   for temporary purposes in the calculation, and a bit saying
   whether or not it is an obstacle cell. */

poly state outgoing_state, incoming_state;
poly unsigned:1 obstacle_cell;
```

```
/* The following declares the actual grid of processors. */

processor fluid_grid[ARRAY_X_SIZE][ARRAY_Y_SIZE];

/* Grid is the C pointer type that corresponds to the above array type. */

typedef processor (*grid)[ARRAY_Y_SIZE];
```

At each time step, instructions are broadcast that tell each cell how to gather data about particles heading in its direction. When the cells poll each of their six neighbors for information, they formulate their own 6-bit incoming state. For example, a cell would ask its East neighbor for its outgoing state bit number 3, and would place the answer in its own incoming state bit number 0. It would then ask its NorthEast neighbor for its outgoing state bit number 4 and would place the answer in its own incoming bit number 1. All cells, in parallel, check the state of all six of their neighboring cells. This extreme data level parallelism allows for a large amount of data to be collected in a small amount of time.

```
/* This code is executed within each processor.  Outgoing state
   bits from six neighbors are gathered and placed within the local
   incoming_state array.  Note the use of a C cast expression
   ((grid)this) to create a self-pointer that has a two-dimensional
   array type suitable for double indexing.  (This code actually is
   oversimplified in that it does not handle the boundary conditions
   for cells on the edge of the grid.  Handling these conditions is
   a bit tedious but conceptually straightforward.) */

poly void get_neighbors() {
    incoming_state.Bit[0] = ((grid)this)[ 1][ 0].outgoing_state.Bit[3];
    incoming_state.Bit[1] = ((grid)this)[ 0][ 1].outgoing_state.Bit[4];
    incoming_state.Bit[2] = ((grid)this)[-1][ 1].outgoing_state.Bit[5];
    incoming_state.Bit[3] = ((grid)this)[-1][ 0].outgoing_state.Bit[0];
    incoming_state.Bit[4] = ((grid)this)[ 0][-1].outgoing_state.Bit[1];
    incoming_state.Bit[5] = ((grid)this)[ 1][-1].outgoing_state.Bit[2];
}
```

Once each cell has determined which particles are entering (by collecting its incoming state), it updates its outgoing state to reflect the particle interactions. First, all cells that have their obstacle-bit turned on are instructed to set their outgoing state to be the same as their incoming state (since particles that hit an obstacle bounce back in the same direction).

Next, patterns are broadcast that correspond to each of the possible 6-bit incoming states, followed by the corresponding 6-bit outgoing state. Each cell compares its incoming state to the pattern being broadcast. When there is a match, the cell updates its outgoing state accordingly. For example, a cell with an incoming state of 011011 would then have an outgoing state of 110110 (refer to figure 3.2d).

```
/* The rule table is indexed by a six-bit incoming-state value
   and contains the corresponding outgoing-state values. */

state rule_table[64];

/* Calculate the new outgoing_state for all cells, based on the
   incoming_state and the obstacle_cell bit. */

poly void update_state {
    if (obstacle_cell)
        outgoing_state.Val = incoming_state.Val;
    else outgoing_state.Val = rule_table[incoming_state.Val].Val;
}
```

It is important to note that this trivial, non-computational, table look-up is the driving force of the whole simulation. The Connection Machine system has replaced all of the mathematical complexity of the Navier-Stokes equations with this small set of bit-comparison operations. The simulation is successful because the system can perform this operation on huge numbers of particles in very short amounts of time. It is an example of the Connection Machine system being easier to program because it supports a much simpler algorithm.

3.4 Interactive Interface

A typical "run" of a fluid flow simulation begins by allowing the user to make several choices. The user typically specifies the average number of particles per cell (density) and the average speed and direction of the particles (velocity). Technically this means that the entire region starts out with particles randomly distributed among the cells (based on the density) and moving in a certain overall direction (based on the average velocity). The user also selects or draws one or more obstacles and places them somewhere in the region being observed. All cells that are part of an obstacle have their obstacle bit set. As the simulation runs, new particles are randomly injected from the edges of the region in order to maintain the selected density and velocity. Once the model is running, each cell's state is continually updated, and average results for regions of cells are displayed.

```
/* This is the main computation loop.  At each time step, each
   cell fetches state from neighbors and updates its own state;
   then the results are displayed. */

poly void fluid_flow() {
    for (;;) {
        get_neighbors();
        update_state();
        display_state();
    }
}

/* Execution begins here. */

void start_fluid_flow() {
    /* Initialization. */
    initialize_rule_table();
    initialize_cell();
    /* Activate all processors in fluid_grid
       and then call the function fluid_flow. */
    [[][]fluid_grid].{ fluid_flow(); }
}
```

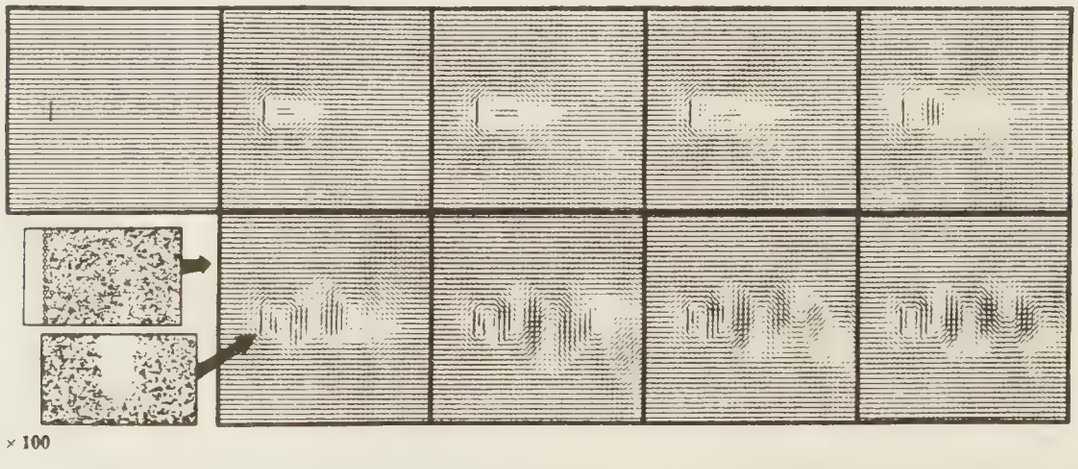


Figure 3.4: *The formation of a fluid flow phenomenon, called a "vortex street," as fluid flows from left to right past a flat plate.*

3.5 Timing and Performance

A production level version of the algorithm described in this chapter has been implemented and extensively tested on the Connection Machine system. The simulation operates on a 4000×4000 grid of cells, typically containing a total of 32 million particles. The Connection Machine system is able to perform one billion cell updates per second. Figure 3.4 shows several displays from a simulation of 100,000 time steps. Each time step includes approximately 70 logical operations per cell; the simulation therefore required a total of 100 trillion (10^{14}) logical operations. The complete simulation took less than 30 minutes. Current results are very competitive with state-of-the-art direct numerical simulations of the full Navier-Stokes equations.

3.6 Summary and Implications

In addition to providing very accurate simulation of fluid behavior, the Connection Machine method for simulating fluid flow allows scientists to continually interact with the model. Any of the user's original choices may be modified during a run of the simulation, without long delays for new results. Since particles are continually moving through the cells, a new density or average velocity may be established by adjusting the particles being randomly injected from the edges. When a new obstacle is added during a run, the obstacle bits in the appropriate cells are set, and those cells begin to reflect particles. Within less than a minute (a few thousand time steps), results based on the new selections become apparent in the displayed flow.

The algorithm for simulating fluid flow on the Connection Machine system is simple. It overcomes problems formerly associated with computer simulations of fluid flow by using a discrete simulation that takes advantage of the Connection Machine system's inherent data level parallelism. During each time step, every particle can move in the direction it is heading, every cell can evaluate its new particles based on collision rules, and every cell can update its state to reflect the direction of the particles it currently contains. The algorithm involves a small number of instructions executed over a large amount of data. Since the Connection Machine system is able to assign a processor to each data element, and to allow all processors to communicate simultaneously, it has provided the computational power required to provide the ideal solution to this applications need.

Chapter 4

Contour Maps from Stereo Images

Human beings have extremely sophisticated and well-developed visual capabilities, which scientists are just now beginning to understand. Since humans are very good at dealing with visual data, graphics and image processing provide an excellent opportunity for creative partnership between people and computers. An example of this partnership is the widespread use of graphical output for computer applications, such as scientific simulations. The computer does what it does best, computing the results and displaying them in a picture or a movie. Researchers do what they do best, using their sophisticated visual system to make qualitative judgements based on the visual information.

In many important computer applications, however, this partnership breaks down. When the flow of visual data is too large, the human visual system makes mistakes. Often this is simply because humans get tired and lose their concentration when faced with very large and monotonous streams of visual data, not because they are trying to extract information too subtle for current computer science to handle.

4.1 Analyzing Aerial Images by Computer

The analysis of detailed aerial images is an area where increased computer processing is highly desirable. Topographers would like to have the computer partially "digest" the visual data first, presenting only the essential properties of the images to the human user. In some cases, they would like to have the computer go even further, drawing abstract conclusions from raw visual data. Scientific progress in image processing and artificial intelligence has recently made this kind of information processing possible. However, conventional computers cannot keep up with the enormous flow of data that these applications present. Consequently, humans are still doing most of the work in these areas. The partnership has broken down because people are doing what the computer should be doing for them.

Data level parallelism is helping to redress this balance. It is ideally suited to the analysis of multiple images and the detection of subtle differences between them. In particular, it is allowing stereo vision algorithms to be applied to terrain analysis in very high volume applications. Stereo vision is the process by which humans are able to take in two slightly different images (from the two eyes) and use the small differences arising from the two different perspectives to determine the distances to the objects in the field of view. Using the same principle, the Connection Machine system is able to analyze two aerial images to determine the terrain elevation and to draw a contour map. Contrary to the apparent ease with which humans can perform this process, it is a subtle and difficult computational problem which no computer has yet solved perfectly. That is why humans are always involved to "coach" the process. The Connection Machine system, with its natural ability to handle large numbers of images and compare them in great detail, can help to drastically reduce the amount of work people must do in this area.

This chapter describes the underlying algorithms for stereo vision on a data level parallel computer, and shows some of the implementation on the Connection Machine system. Many detailed elements of an actual production system, such as straightening out misaligned images and displaying intermediate results, have been omitted in order to focus on the underlying algorithms. See references [2,3,5,11,12,13] for more information on machine vision and the stereo matching problem.

4.2 Seeing in Stereo

Images are very large, inherently parallel data structures. Therefore the processing of images is an application that is ideally suited for data level parallelism. An image is stored as an array of *picture elements*, or *pixels*. An image with 256 pixels in the vertical dimension and 256 in the horizontal dimension has a total of 65,536 data elements. More detailed images, with 1024 by 1024 pixels, have more than a million data elements. For black and white images, the value stored in each of the pixels is the intensity of light at that point, ranging from pure white through various shades of gray to pure black. (Pixels in color images contain information describing the hue and saturation as well as the brightness.) The contour mapping problem is one of extracting terrain *elevation* information from images that, upon first inspection, contain only information about terrain *brightness* at each pixel.

The term *stereo* means "dealing with three dimensions." *Stereo vision* is "the ability to see in three dimensions." Humans and many animals have the remarkable ability to take in two images, obtained from slightly different perspectives—one from each eye—and fuse them to perceive a three-dimensional world. The difference in perspective causes objects to appear in slightly different places in the two images. The amount of positional difference is related to the distance of the object from the viewer.

Because stereo vision occurs automatically in humans, we tend to be unconscious of the process. A simple demonstration serves as a reminder. Hold a pencil in front of a piece of paper and fix your gaze on the paper. Start to alternately close one eye and then the other, then slowly move the pencil toward your face. Keep the paper stationary and your gaze fixed on the paper while you move the pencil. The paper always seems to shift back and forth by the same small amount, but the closer the pencil moves to you, the more it jumps in position between the two views.

The two images used in a stereo vision system are called a "stereo pair." Figures 4.1 and 4.2 give an example. Figure 4.1 shows a model of some terrain, as seen from an oblique angle. Figure 4.2 shows a stereo pair obtained from directly above the terrain. Figure 4.2 can produce a vivid sensation of depth when observed with an appropriate stereo viewing apparatus.

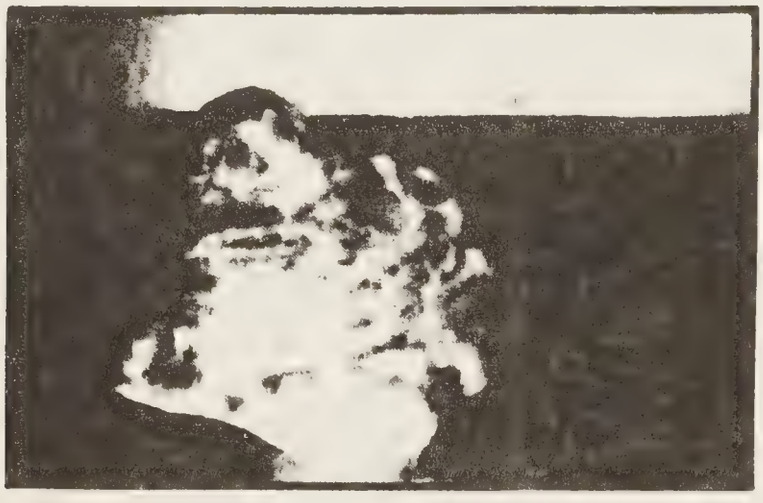


Figure 4.1: *An oblique view of a terrain model used in a demonstration of the contour mapping algorithm.*

4.3 Finding the Same Object in Both Images

Individual pixels within an image are not reliable indicators of objects. Two pixels, one in each image, can have the same brightness value without being part of the same object. Features larger than individual pixels must be found. The "edges" between areas of different intensities make up an effective set of such features. An edge is a line, usually a crooked line, along the boundary between two areas of the image that have different intensity. Instead of trying to match pixels based on their intensity, the algorithms match them based on the *shape of nearby edges.* The shape of edges is usually much more strongly related to

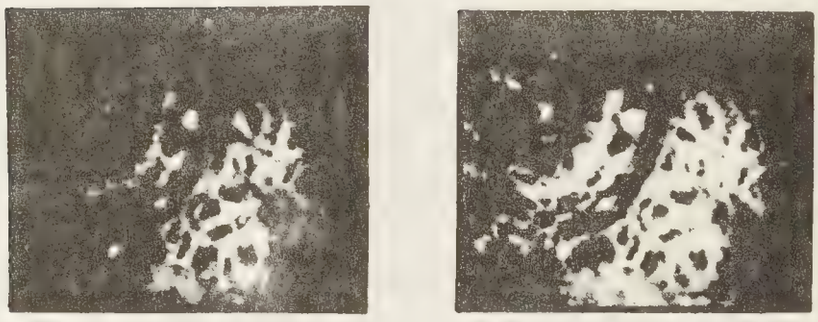


Figure 4.2: *A stereo pair of the terrain in Figure 4.1, obtained from directly above the terrain.*

distinct objects than the simple brightness value.

Figure 4.3 shows an example of edges. These edges were derived from the stereo pair in Figure 4.2.

The process of finding edges falls into the category of image computations called "local neighborhood operations." Individual pixels are classified based on characteristics of a group, or neighborhood, of nearby pixels. Edges are found by having each pixel determine whether the brightness of nearby pixels on one side of it is very different from the brightness of nearby pixels on the other side. This will be the case only for pixels that pass this test: *they must lie between two image regions that are similar within themselves but different from each other.* These *edge pixels* are detected by examining the local neighborhood of every pixel *in parallel*, and storing the ones that pass the test in an array. Typically, only 10 to 20 percent of the pixels in an image get classified as edge pixels.

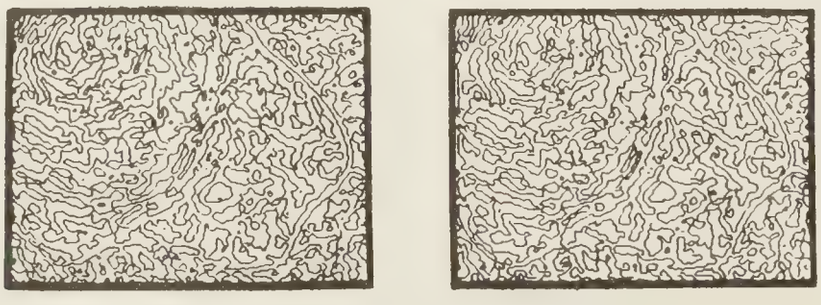


Figure 4.3: *An example of edges. These edges were derived from the stereo pair shown in Figure 4.2. They delineate the boundaries between areas of different intensity.*

4.4 Matching Edges

Even though edges are much more closely tied to objects than simple brightness values, there is still a great deal of work involved in deciding whether an edge in one image corresponds to a particular edge in the other image. Real images suffer from distortions due to several sources. Distortions include random fluctuations or "noise" introduced in the electronic imaging process, relative misalignment between the cameras, and irregular illumination. In addition to these effects, which tend to blur the distinction between edges that match and those that do not, there is a "bad luck" factor: an object or surface marking in one image very often just happens to look like several markings in the other image. For these reasons, the final choice of matches, and therefore the correct positional difference, is always somewhat ambiguous.

If the detection of edges were a perfect process, deciding which positional difference is best for each pixel would be simple. A local neighborhood of edges would align exactly at one relative shift and very little at all the others. Because of the imperfections described above, however, such a high level of precision is impossible. Every neighborhood of edges in one image matches to some extent with many neighborhoods in the other image. The competition is usually very close.

4.5 Measuring Alignment Quality

To resolve the competition, the Connection Machine algorithms hold one of the images stationary and "slide" the other one over it horizontally one pixel at a time. Each time the moving image is slid one more pixel's distance, all the stationary pixels compare themselves to the pixels to which they now correspond in the slid image. They record the presence or absence of an edge alignment in a table in their own memory. Typically, the maximum shift between two images is 30 pixels, so a table of 30 alignment matches is created in the memory of each stationary pixel's processor.

This sliding procedure, using the edges from Figure 4.3, is illustrated in Figure 4.4. Each of the 16 images shows an alignment table entry for each pixel. Black pixels indicate positive alignment table entries, i.e., "match-ups" between the stationary and the sliding images. For example, the 7th image shows alignment-table-slot 7 in each pixel. Thus every black pixel in image 7 corresponds to a match-up between stationary and sliding edges when the relative shift was 7 pixels.

The resulting alignment tables generally show several spurious matches, but also one or two solid ones where the local neighborhood of edges lined up very tightly. When this happens at a pixel, it is a signal that the correct shift (the correct positional difference) for that pixel has been found.

Figure 4.4: *An illustration of the sliding process. Each of these images shows the contents of an alignment-table-slot in each pixel. The Nth image shows slot N in every pixel's alignment table. The dark areas are regions of good alignment, i.e., areas where the same alignment-table-slot is filled in many pixels.*

As in the edge detection process, the alignment quality of every shift position in the alignment table is measured by a local neighborhood operation. In this case, the operation is the following: for each shift position, each pixel processor counts and records the number of matching edge pixels in a small neighborhood around itself. This count or "score" will be high for pixels whose nearby edges are tightly aligned with the edges in the other image *at the same position but displaced by the shift.*

The best shift for a given pixel is determined by comparing the alignment scores at every position in its alignment table. *The shift that has the highest score is chosen as the correct shift for the pixel.* This process takes place in parallel for all pixels; in this way a shift is determined for each pixel.

Areas of tight alignment are clearly visible in Figure 4.4. For example, the small shifts (1 through 4) are tightly aligned over low terrain (refer to Figure 4.1), and the large shifts (13 through 16) are tightly aligned over high terrain. Match-ups in these areas will get high alignment scores because they lie amidst many other match-ups.

4.6 Drawing Contour Maps

The processing described so far yields the shift (or elevation) for every pixel that is part of an edge. These pixels form a "web" of heights that approximates the shape of the terrain, but is not yet smooth and continuous. It is full of holes (where non-edge pixels were) which must be filled in by interpolation.

Interpolation is accomplished by another local neighborhood operation. Each pixel that is not on the web takes on a new elevation which is the average elevation of the pixels in a small neighborhood around it. The neighborhood includes the four pixels above, below, to the left and to the right of the pixel. The pixels that make up the web maintain their original elevations; only the pixels in the holes change their values. This process is repeated or "iterated" a few hundred times.

Pixels that lie in the middle of holes in the web have zero elevation. Therefore, when they become the average of their neighbors, which also have zero elevation, their elevation does not change. However, pixels that lie near the edges of holes in the web have neighbors whose elevation is nonzero. Therefore, when they become the average of their neighbors, they jump to a nonzero elevation. On the next iteration, these new nonzero pixels influence their neighbors, in turn creating new nonzero elevations. Gradually, after a few hundred iterations, the pixels on the web—which remain unchanged throughout the process—"spread" their elevations across the holes in the web, filling it in to create a smooth, continuous surface from which a contour map may be drawn. An example of a contour map is shown in Figure 4.5.

Figure 4.5: *A contour map of the terrain model shown in Figures 4.1 and 2, computed on the Connection Machine system.*

4.7 Finding Edges on the Connection Machine System

A pixel is classified as an edge pixel if it lies between two image regions that are similar within themselves but different from each other. This is the program that performs the edge classification operation.

```
(*defun find-edges-between-left-and-right!! (brightness-pvar threshold)
  (*let* ((average-brightness-on-the-left
           (/!! (+!! (pref-grid-relative!! brightness-pvar (!! -1) (!! -1))
                     (pref-grid-relative!! brightness-pvar (!! -1) (!! 0))
                     (pref-grid-relative!! brightness-pvar (!! -1) (!! 1)))
                (!! 3.0)))
          (average-brightness-on-the-right
           (/!! (+!! (pref-grid-relative!! brightness-pvar (!! 1) (!! -1))
                     (pref-grid-relative!! brightness-pvar (!! 1) (!! 0))
                     (pref-grid-relative!! brightness-pvar (!! 1) (!! 1)))
                (!! 3.0)))
          (average-brightness-overall
           (/!! (+!! average-brightness-on-the-left
                     average-brightness-on-the-right)
```

```
                (!! 2.0))))
    (if!! (>!! (absolute-value!! (-!! average-brightness-on-the-left
                                       average-brightness-on-the-right))
               (*!! (!! threshold) average-brightness--overall))
          (!! 1)
          (!! 0))))
```

The preceding program sequence calculates the average brightness in a small region to the left (i.e., with relative x-coordinate -1, and relative y-coordinates -1, 0, and 1) and the average brightness in a small region on the right side (with relative x-coordinate 1) of each pixel. If, at any particular pixel, the difference between these averages is greater than the specified threshold, then the pixel is marked with a one, meaning that it is an edge pixel. Otherwise it is marked with a 0. The threshold is multiplied by the overall average brightness, a process called "normalization." With normalization, the threshold adapts to the image, becoming small in regions where the image is generally dark, and large where the image is generally bright.

Since this program compares regions on the left and right sides of a pixel, it works only for edges that are more or less vertical. It is easy to write a program that finds horizontal edges by having it compare small regions on the top and bottom of a pixel, in the same way that this program compares regions on the left and right. The same could be done edges in both diagonal directions. The four programs may then be combined to find *all* edges in the following way:

```
(*defun find-all-edges!! (brightness-pvar threshold)
  (if!! (or!! (=! (!! 1) (find-edges-between-left-and-right!!
                            brightness-pvar threshold))
              (=! (!! 1) (find-edges-between-above-and-below!!
                            brightness-pvar threshold))
              (=! (!! 1) (find-edges-between-upper-left-and-lower-right!!
                            brightness-pvar threshold))
              (=! (!! 1) (find-edges-between-lower-left-and-upper-right!!
                            brightness-pvar threshold)))
        (!! 1)
        (!! 0)))
```

4.8 Matching Edges on the Connection Machine System

The following program sequence implements the sliding procedure described above. One of the edge images is held stationary and the other edge image is moved across it horizontally,

one pixel at a time. At each relative shift (1, 2, ..., 30), each processor records whether an edge match has been found in the sliding image. This information is stored in a pvar that represents one of the alignment tables discussed above. All of the alignment tables are stored in the Connection Machine memory at the same time.

```
(defvar *array-of-pvars-holding-matches-at-each-shift* (make-array 30))
  ;;; This is just a regular Lisp array, but each element of this
  ;;; array will be a pvar.  Notice that we'll try to find positional
  ;;; differences of up to 30 pixels.  (Note: each one of the pvars
  ;;; in this array will hold an "alignment-table-slot" for every pixel,
  ;;; as discussed in the text).

(*defun fillup-pvars-wherever-edges-align (left-edges right-edges)
  ;; This program records the edge-pixel match-ups at every shift;
  ;; that is, this program creates "match-up images," as shown in
  ;; Figure 4.4.
  (dotimes (i 30)
    (aset (if!! (=!! left-edges
                     (pref-grid-relative!! right-edges (!! i) (!! 0))
                     )      ; ^This PREF-GRID-RELATIVE!! accomplishes
                (!! 1)      ; the "sliding" process.
                (!! 0))
          *array-of-pvars-holding-matches-at-each-shift*
          i)))
```

The next step in the process is to decide at each pixel position which shift produced the best match-up. Most locations will contain a somewhat random pattern of match-up pixels. However, at some locations, the local neighborhood of match-ups will be very dense and regular, indicating that the shift responsible for that match-up image is probably the correct shift for that neighborhood.

The following *Lisp program measures the density or alignment quality of every neighborhood. It does so by counting the number of 1's (match-ups) in a square around each pixel. The counting process is accomplished in parallel, for all pixels at once, on the Connection Machine system.

```
;;; The square for each pixel is to be centered on that pixel.
;;; Because a DOTIMES loop always produces values starting at zero,
;;; it is necessary to subtract one-half the width of the square
;;; from the loop variable in order to get relative indexes that
```

```
;;; are centered on zero.

(*defun add-up-all-pixels-in-a-square (pvar width-of-square)
  (let ((one-half-the-square-width (/ width-of-square 2)))
    (*let ((total (!! 0)))
      (dotimes (relative-x width-of-square)
        (dotimes (relative-y width-of-square)
          (*set total
                (+!! total
                     (pref-grid-relative!!
                        pvar
                        (- relative-x one-half-the-square-width)
                        (- relative-y one-half-the-square-width))))))
      total)))
```

At this point, it is a simple matter to record the alignment quality or score for every pixel.

```
(defvar *array-of-pvars-holding-scores-at-each-shift* (make-array 30))
  ;;; Another Lisp array holding *Lisp pvars.
```

The next step is to fill all the elements of the Lisp array with *Lisp pvars. The Nth element of the Lisp array holds a pvar containing the scores, or alignment qualities, of all the matches that occurred when the edge images were shifted by N pixels relative to each other. (Note that this program records scores only at locations where match-ups occurred. Other locations have no score, which reflects our original intention of matching *edges*, not the holes between them.)

```
(*defun fillup-pvars-with-match-scores (width-of-square)
  ;; WIDTH-OF-SQUARE will typically be 21.
  (dotimes (i 30)
    (*let ((sum-of-all-nearby-pixels
             (add-up-all-pixels-in-a-square
               (aref *array-of-pvars-holding-matches-at-each-shift* i)
               width-of-square)))
      (*if (=!! (aref *array-of-pvars-holding-matches-at-each-shift* i)
                (!! 1)) ;;; Record a score wherever there was a match-up.
           (*set sum-of-all-nearby-pixels
                 *array-of-pvars-holding-scores-at-each-shift*
                 i)))))
```

Now that the score for every match-up has been recorded, there is only one more step required to establish which of the match-ups is correct. The following *Lisp program loops through all the shifts, keeping track of the best score at each pixel. The shift that produced the best score at each pixel is recorded as the "winning shift."

```
;;; This function computes the web of known shifts.  Recall that
;;; the shift at each pixel corresponds directly to the elevation.

(*defun find-the-shifts-of-the-highest-scoring-matches ()
  (*let ((best-scores (!! 0))
         (winning-shifts (!! 0)))
    ;; The following DOTIMES loop makes sure that each
    ;; pixel in the BEST-SCORES pvar contains the maximum
    ;; score found at any shift.
    (dotimes (i 30)
      (*if (>!! (aref *array-of-pvars-holding-scores-at-each-shift* i)
                best-scores)
           (*set best-scores
                 (aref *array-of-pvars-holding-scores-at-each-shift* i))))
    ;; The following DOTIMES loop records a "winning"
    ;; shift at every pixel whose score is the best.
    (dotimes (i 30)
      (*if (=!! (aref *array-of-pvars-holding-scores-at-each-shift* i)
                best-scores)
           (*set winning-shifts (!! (1+ i)))))
    winning-shifts))
```

4.9 Drawing Contours on the Connection Machine System

A contour map cannot be constructed without a smooth, continuous surface on which to draw the lines. All of the processing so far has produces a web of known elevations (returned by the last *Lisp function above). Interpolation across the holes in the web produces a continuous surface.

```
(*defun fill-in-web-holes (web-of-known-elevations times-to-repeat)
  ;; Each time through the loop, every pixel not on the web (i.e.,
  ;; every pixel that is not zero to begin with) takes on the
  ;; average elevation of its four neighbors.  Therefore, the web
  ;; pixels gradually "spread" their elevations across the holes,
  ;; while they themselves remain unchanged.
  (dotimes (i times-to-repeat)
    (*let ((not-fixed (zerop!! web-of-known-elevations)))
      (*if not-fixed
         (*set web-of-known-elevations
               (/! (+!! (pref-grid-relative!!
                         web-of-known-elevations
                         (!! 1) (!! 0))          ;Neighbor to the right
                        (pref-grid-relative!!
                         web-of-known-elevations
                         (!! 0) (!! 1))          ;Neighbor above
                        (pref-grid-relative!!
                         web-of-known-elevations
                         (!! -1) (!! 0))         ;Neighbor to the left
                        (pref-grid-relative!!
                         web-of-known-elevations
                         (!! 0) (!! -1)))        ;Neighbor below
                   (!! 4))))))
   web-of-known-elevations) ;;; this is now a more or less smooth surface.
```

The following code takes the smoothed-out web and constructs a contour map in the form of a plane of black-and-white pixels suitable for display on a graphics device.

```
(*defun draw-contour-map (number-of-contour-lines
                          pvar-of-smooth-continuous-elevations)
  ;; The idea is to divide the whole range of elevations into
  ;; a number of intervals, then to draw a contour line at every
  ;; interval.
  (let* ((max-elevation (*max pvar-of-smooth-continuous-elevations))
         (min-elevation (*min pvar-of-smooth-continuous-elevations))
         (range-of-elevations (- max-elevation min-elevation))
         (contour-line-interval (/ range-of-elevations
                                   number-of-contour-lines)))
```

```
  ;; Now the variable CONTOUR-LINE-INTERVAL tells us how many
  ;; elevations, or shifts, to skip between contour lines.
 (if!! (zerop!!
         (mod!! (-!! pvar-of-smooth-continuous-elevations
                     (!! min-elevation))
                (!! contour-line-interval)))
       (!! 1)    ;; This IF!! draws all the elevation contours
       (!! 0)))) ;; at once, returning a bit map suitable for
                 ;; for immediate display.
```

4.10 Timing and Performance

A production level version of the contour mapping algorithm described in this chapter has been implemented and extensively tested on the Connection Machine system. Parameters such as the size of the images and the range of positional differences ("shifts") are variable, depending on the application. A typical program run processes images containing 512×512 (262,144) pixels, while allowing for positional differences from 0 to 30 pixels. In such a mode, the Connection Machine system performs approximately two billion (2×10^9) operations during the most time-consuming phase of the algorithm, the so-called "inner loop," in which the match-ups are detected and their alignment quality is measured. This inner loop is executed in less than two seconds.

4.11 Summary and Implications

Contour mapping using stereo vision is an example of an image processing application that is sophisticated and computationally expensive. The Connection Machine system, because it readily accommodates itself to the inherently parallel structure of image data, made it easy to conceptualize and to program the contour mapping algorithm. The simplicity and brevity of the programs shown above is evidence of this natural fit.

The raw speed of the Connection Machine system is as valuable as its architecture. The system can extract elevation information from large amounts of visual data at very high rates. This speed allows scientists and engineers who are developing new techniques in computer vision to try their ideas "on the fly." A short turnaround time for experimenting with new ideas is essential for the rapid development of the field of computer vision. The effects of various program modifications are realized almost instantaneously. The system's computational power is a valuable aid in the design and implementation of sophisticated algorithms.

Chapter 5

The C* Programming Language

C* (pronounced *see star*) is a simple extension to the C programming language [6,10] that exploits the power of the Connection Machine architecture. C* is (almost) a strict extension of C; any valid C program, if it avoids the use of a small number of C* reserved words, is also a valid C* program. A few new features of the language serve to indicate where data is stored and which operations are executed in parallel in the Connection Machine network.

5.1 C* Extensions

In order to indicate whether a variable is located on the host or in the Connection Machine memory, two storage class identifiers `mono` and `poly` have been included in C*.

```
mono int x;     /* x resides in the host memory */
poly int y;     /* y resides in the Connection Machine memory */
```

The modifier `poly` declares variables present in all processors.

The majority of parallel code is standard C code. Parallel functions are simply distinguished by the identifier `poly`. It is a mark of the general-purpose nature of the Connection Machine architecture that the full C language is available for programming the processors of the Connection Machine system. Likewise, it is a mark of the simplicity of the architecture that the C language suffices for this task. In fact, no new language features need to be introduced in order to perform parallel control flow, interprocessor communication, and memory allocation. The real power of C* comes from the natural parallelization of familiar constructs of C.

5.1.1 Parallel Control Flow

Inside of a parallel function, the normal C control-flow statements, such as `if` and `while`, work as expected. This is perhaps unexpected to someone experienced with other parallel languages. For example, an `if` statement may have a conditional expression whose value is different in different processors:

```
poly salary;
...
if (salary <= 0)
    salary = fixup_salary();
```

It would clearly be an error for *all* processors to make the call to `fixup_salary`. The way C* handles such a statement is to reduce the active set of processors, by temporarily inactivating all those whose `salary` variables are positive. The body of the `if` statement is run, and then the original active set is restored. Such conditional statements can be nested to any degree.

The `while` statement can also operate in parallel. At each evaluation of the loop's conditional expression, more processors can drop out of the active set; they stay inactive until the loop is finished. Finally, when all processors are finished with the loop, the statement is done, and the original active set is restored. For example:

```
poly resumes_to_read;
...
while (resumes_to_read > 0) {
    /* Read ten resumes at a time. */
    resumes_to_read -= 10;
    ...
}
```

In this case, all processors with `resumes_to_read` between 1 and 10 execute the loop body exactly once.

All other standard C control constructs are handled in similar ways in C*; even *goto* is accommodated. The program behaves as if the standard C code were running separately in each processor, with processors that are doing the same thing doing it at the same time.

5.1.2 The Selection Statement

In order to execute code in a selected set of processors, an additional statement called the *selection statement* is included in C*. Selection statements may be used within any C* function. The selection statement has the form:

```
[selector].statement
```

The selector indicates a set of processors. These are activated, and the `statement` is executed within those processors. For example, given the following declaration,

```
processor managers[100];
```

the following statement

```
[[100]managers].{ salary *= 1.06; }
```

or, more simply,

```
[[]managers].{ salary *= 1.06; }
```

selects all 100 of the `managers`, and gives them a six percent raise. The code:

```
[[50]managers].{ salary *= 1.11; }
```

gives the first 50 an eleven percent raise, while this:

```
[managers[0],managers[2]].{ salary -= 1000; }
```

singles out the first and third managers for a pay cut. (More complicated forms of selection are also available.)

5.1.3 Computation of Parallel Expressions

C* extends the meaning of C expressions to parallel computations by means of two simple rules. The first rule says that if a single value (typically of storage class `mono`) is combined with a parallel value (of class `poly`), the single value is first replicated to produce a `poly` value. (In hardware terms, the single value is *broadcast* to all relevant processors.) For example, in the expression `(salary > 20000)`, the single value 20000 is replicated to match the parallel variable `salary`. This rule is an addition to the rules of "usual conversions" in plain C.

The second rule says that an operation on a parallel value (or values) must be processed *as if* only a single operation were executed at a time, in some serial order. In the expression `(salary > 20000)` it is *as if* we took first one `salary` value and compared it to 20000, then another, and so on, doing the comparisons one at a time.

Fortunately, we can analyze the > operation and determine that doing all the comparisons at once will produce the same result, because doing so will not affect the outcome. This is hardly surprising, and it is exactly the effect we want anyway, so why do we have the "as if serial" rule at all? It is because some operators *do* have side effects: assignment operators. Consider the expression

```
total_payroll += salary;
```

Now `total_payroll` is a single value (what in C is called an *lvalue*, because it occurs on the left side of an assignment). By the first rule it is replicated. We then have many assignments to perform, one for each value in the parallel value `salary`:

```
total_payroll += salary_1;
total_payroll += salary_2;
total_payroll += salary_3;
        .
        .
        .
```

The second rule guarantees that the program behaves *as if* all of these assignments were performed in some serial order. *Which* order does not matter; the result is the same. The point is that if these assignments were executed in parallel some updates might be lost; but C* guarantees that *all* the `salary` values will be correctly added into `total_payroll`. (Doing this efficiently is handled by the C* implementor.)

A C assignment operator may be used as a unary operator in C* to reduce a parallel value to a single result that may be further operated upon. For example,

```
(+= salary)
```

adds up the salaries for all persons for which processors are active, and

```
(+= salary)/(+= ((poly) 1)))
```

computes the average of all salaries because the expression

```
((poly) 1)
```

makes a 1 for every active processor and

```
(+= ((poly) 1)))
```

adds up all the 1's, thereby counting all the active processors.

In C*, "<>" is the "minimum" operator and "><" is the "maximum" operator. The expression "`a >< b`" means the same as "`(a > b) ? a : b`". The assignment operators `<>=` and `><=` are also defined: "`a <>= b`" assigns `b` to `a` if `b` is less than `a`. The expression `(><= salary)` finds the largest salary, and `(<>= salary)` finds the smallest salary.

5.1.4 Data Movement

C* has no language extensions to handle data movement or interprocessor communication *per se*. Instead, the normal C operations are used; the Connection Machine architecture allows random access to the desired datum, wherever it is in the system.

Within the code of a `poly` function, the keyword **`this`** is a C* reserved word whose value is a pointer to the currently executing processor. This value is sometimes called the *self-pointer*. If many processors are executing, each will have its own self-pointer. References to the processor's variables implicitly refer to the self-pointer: saying **`salary`** is the same as saying **`this->salary`**. Explicit references to **`this`** are useful for accessing the memory of neighboring processors through indexing.

The key point is that any processor may contain a pointer to data in the memory of any other processor, and access through that pointer is supported by the Connection Machine router. All interprocessor communication can therefore be expressed in C* merely by the usual explicit and implicit pointer indirection mechanisms. For example, to increment a neighbor's `salary` field, and then decrement one's own based on the result, the following code might be used:

```
this[1].salary += 1000;
salary -= this[1].salary * .10;
```

Similar expressions can also be used to broadcast data throughout the system, to transfer data between the host and Connection Machine processing network, or to collect data from many sources into one location.

5.2 Summary

The C* language is a version of the standard C language suitable for programming the Connection Machine system. Because of the simplicity and power of the Connection Machine architecture, C* itself is a simple yet powerful extension of C. The Connection Machine memory is treated as a large section of host-accessible memory with active objects stored in it. Because standard C is already excellent at manipulating structures, pointers, and the like, relatively few new language features are needed to deal with the Connection Machine architecture. All the familiar C language constructs acquire the power of parallelism easily and naturally.

Chapter 6

The *Lisp Programming Language

*Lisp (pronounced *star lisp*), is an extension of Common Lisp [9], a standard dialect of Lisp that is found on a variety of computer systems. Lisp has many features that are common to most programming languages, but its unusual structure and syntax make the programs a bit difficult to read for someone who has mainly had experience with block structured languages such as FORTRAN or C.

This chapter covers both Lisp and *Lisp in sufficient depth to make it possible to understand the program examples in this book. See references [9,15,16] for a deeper understanding of the Lisp language and its structure.

6.1 Fundamentals of Lisp

What most people remember about Lisp is that it uses lots of parentheses. And it is true—Lisp does. But it is not necessary to understand the full implications of the parentheses to understand the sample programs. Roughly, in a Lisp expression the first thing that comes after the open parenthesis is the function name, and after that are the arguments. So `(+ 7 A)` would call the function `+`, which adds 7 and the value of the variable `A`, and returns the result.

Lisp function calls can be nested as they can in other languages. For example:

```
(* 5 (+ 1 2 3))
```

would first add together 1, 2, and 3, and then multiply the result by 5, giving 30.

Most Lisp programs are indented to help reveal their structure and to show how many levels deep parentheses have been nested. Expert Lisp programmers keep their code properly indented, and rely on the indentation as much as the parentheses when reading code.

6.1.1 Lisp Functions

Functions are the program building blocks of Lisp. Unlike many other programming languages, Lisp does not have a main program followed by a series of functions. In Lisp everything is a function, and programs are executed by invoking those functions from an interactive Lisp interpreter.

The Lisp function-defining operation is called `DEFUN`. The first argument to `DEFUN` is the name of the function that is being defined, the second a list of its arguments; these are followed by the operations to be performed. For example:

```
(defun add-three (x) (+ x 3))
```

defines a function named `add-three` that takes one argument named x, and the operation that is performed by the function is `(+ x 3)`.

6.1.2 Variables

It is not necessary in Lisp to predefine variables, but it is often done for clarity. The mechanism is straightforward:

```
(defvar a 25)
```

defines a variable named `a` with an initial value of 25. Variables defined with `defvar` are global variables that can be accessed by any function at any time.

Temporary variables are defined in Lisp with the `let` operation, which takes a list of variable-value pairs, and is followed by a sequence of operations to be performed. For example,

```
(let ((temporary 25)
      (x 49))
  (print (+ temporary x))
  (print (* temporary x)))
```

allocates two temporary variables `temporary` and x, assigns them the values 25 and 49 respectively, prints their sum and product, and then deallocates them when the `let` is exited.

Variables have their value set with the `setq` function which takes as its arguments a variable name and a value. So

```
(setq b 34.5)
```

sets the variable `b` to 34.5.

6.1.3 Program Control Structure

The `if` construct is a simple method for conditionally controlling the flow of a program; it is used in several places in the example programs. It takes a test clause, an expression to evaluate if the result of evaluating the test clause is true, and, optionally, an expression to evaluate if the result is false. The following simple example shows how `if` is used.

```
(if (= a 10)
  (print "a is 10")
  (print "a is not 10"))
```

Several of the examples use `dotimes`, a facility for executing a series of expressions a specified number of times. As an example,

```
(dotimes (j 10)
  (print j))
```

prints the integers from 0 to 9.

6.2 *Lisp Extensions

A *Lisp program looks much like an ordinary Lisp program. The biggest distinction is that *Lisp operations manipulate data stored in the Connection Machine hardware, while Lisp operates exclusively on the host processor. There are no instructions stored in the Connection Machine processors; instructions are generated from the *Lisp program and broadcast to the Connection Machine system.

The names of most *Lisp functions either begin with an "*" or end in "!!" (meant to look like two parallel lines, and pronounced *bang bang*) which means that they perform operations on parallel variables. This is only a naming convention and does nothing but distinguish functions that work with the Connection Machine system and parallel variables from functions that don't. User programs may also follow the convention, but it is not a requirement.

This section describes enough *Lisp to make the example programs understandable. As part of that, it is first necessary to describe a few of the fundamental features of the Connection Machine system.

6.2.1 Processors

A *processor* is the entity that operates on data in parallel. Each processor has a unique address that allows it to be directly accessed. The address is made up of one or more numbers depending how many dimensions the Connection Machine hardware is simulating. A

one dimensional machine would take one number as an address, a two dimensional machine two numbers, etc. *Lisp has instructions that can directly access data in the Connection Machine processors via these addresses.

6.2.2 Parallel Variables

The parallel variable mechanism is one of the key programming differences between *Lisp and sequential programming languages. A thorough understanding of what parallel variables are and how they work is crucial to understanding the example *Lisp programs in this document.

On a serial machine a variable may have only one value at a time. On the Connection Machine system a parallel *Lisp variable has as many values as there are processors. Descriptors for parallel variables, or *pvars*, reside on the host computer, and the values of those parallel variables are in the Connection Machine memory.

The *Lisp expression for defining a pvar is similar to the Lisp mechanism for allocating a variable. The expression

```
(*defvar b (!! 5))
```

defines a pvar named `b` which has a value of 5 on every processor in the machine. The function `*defvar` is the parallel version of Lisp's `defvar`. The expression

```
(!! 5)
```

is the part of the `defvar` that actually does the allocation of a field with a value of 5 in every Connection Machine processor.

Values are retrieved from processors with the `pref` function. For example,

```
(pref b 7)
```

would return the value of pvar `b` in processor 7. Setting a value in a processor is accomplished with the Lisp `setf` function.

```
(setf (pref b 3) 10)
```

would set the value of pvar b to 10 in processor 3. The first argument to `setf` describes how to access the field that is going to be altered and the second argument is the new value of the field.

The following series of *Lisp expressions show in some detail how to allocate and use pvars.

First define some pvars:

```
(*defvar a)
```

```
(*defvar b (!! 5) "This is a documentation string.")
(*defvar c (!! -2.67))
(*defvar d t!!)
(*defvar e (1+!! (self-address!!)))
```

These statements created five pvars. The last four have been initialized with specific values: b is a Lisp symbol that has as a value a pvar whose contents is the integer 5 in each processor, c contains the floating point number -2.67 in each processor, d contains the boolean value true in each processor, and e contains the address of the next higher processor. The function `self-address` is a function that returns a pvar which contains the address of the selected processor.

Now read some of the values using `pref`.

```
(pref c 0)
```

returns the lisp value -2.67 since that is what is contained in pvar c in processor 0.

```
(pref d 365)
```

returns the lisp value t since that is what is contained in pvar d in processor 0.

Now do some arithmetic on these pvars:

```
(*set a (+!! b c))
```

will set the contents of pvar a to be the sum of the contents of pvar b and pvar c. Notice that c contains floating-point values. The integers contained in b are converted to floating-point numbers and the result in a will be floating point as well. Expressions can be nested:

```
(*set a (-!! b (*!! a (!! 2))))
```

This expression sets a to the difference of b and twice a. This simple expression could cause thousands of such operations to go on simultaneously! The expression `(!! 2)` returns a pvar that is 2 in all processors.

This point is important. The expression

```
(+!! a 2)
```

is an incorrect *Lisp expression. The variable a is a pvar, whose values are stored on the Connection Machine system, while the integer 2 is a Lisp object stored on the front end system. It is necessary to convert the 2 to a parallel value before doing any parallel computation.

6.2.3 Accessing Pvars Relative to a Grid

Two of the example programs, fluid flow and stereo matching, make heavy use of the Connection Machine system's grid mechanism, which facilitates communications between processors for problems with two-dimensional data structures. For example say `image` was a pvar containing a two-dimensional image. The following expression would shift the entire image over by one pixel in the x direction and place the result in `shifted-image`:

```
(*set shifted-image (pref-grid-relative!! image (!! 1) (!! 0)))
```

in this example the `(!! 1)` specifies that there is a shift of 1 in the x-dimension, and the `(!! 0)` specifies that there is no shift in the y-dimension.

6.2.4 Selection

In *Lisp it is possible to do an operation in a selected subset of all processors. The *Lisp function `*when` is used to do that selection. For example:

```
(*when (=!! a (!! 5))
  (*set a (+!! (!! 2))))
```

adds two to `a` in all processors in which `a` has a value of 5.

6.2.5 *Lisp Programs

*Lisp programs are defined in much the same way that Lisp functions are defined. The main difference is that `*defun` is used instead of `defun` to define functions that either take a parallel variable as an argument or return a parallel variable as a result.

6.3 Summary

*Lisp is a simple extension to Common Lisp that integrates the Connection Machine system into an ordinary serial programming environment. For someone familiar with Lisp, the essentials of *Lisp can be learned and put to productive use within a few hours.

Chapter 7

The Connection Machine System

The Connection Machine system from Thinking Machines Corporation is the first computer to implement data level parallelism in a general purpose way. It combines a very large number of processors with the communications capability necessary to match data topologies exactly. This chapter describes the hardware implementation of the Connection Machine system.

7.1 Connection Machine Internal Structure

As described in Chapter 1, the Connection Machine system operates by receiving a stream of instructions from its front end computer. A microcontroller receives the instructions, expands each of them into a series of machine instructions, then broadcasts the machine instructions, one at a time, to all processors at once. The instructions coming in from the front end are referred to as "macro-instructions." The instructions broadcast to the individual processors are called "nano-instructions." Macro-instructions are similar to assembly language instructions on a conventional machine. They are the instruction codes produced by the Connection Machine language processors. In the sections that follow, names of macro-instructions appear in italics.

The Connection Machine system includes 65,536 physical processors, but may be configured for a much larger number of logical processors by means of the *cold-boot* command. *Cold-boot* takes two arguments that allow a two-dimensional array of virtual processors per physical processor. *Cold-boot*(4,4), for example, sets up the machine in the million-processor mode (or, more precisely, the 1,048,576 processor mode) because each of the 65,536 processors will simulate 16 (4×4) virtual processors. The same number of virtual processors could be established by the command *cold-boot* (16, 1). Since virtual processors are so commonly used, they are referred to simply as "processors". Where it is necessary to refer to one of

the 65,536 hardware processors, the term "physical processor" is used.

Each physical processor has 4096 bits of memory, totalling 32 megabytes for the machine as a whole. In the million-processor mode, each processor has 256 bits of memory. Memory is divided into a data area and a stack area, with the layout being the same in each processor. A single, system-wide register, the stack limit, defines the boundary between stack space and data space. The stack pointer is also a system-wide register. The stacks in all processors act in unison.

Memory is bit-addressable; all data fields are of arbitrary length. For numeric computing there are three standard formats: unsigned-integer, signed-integer, and floating-point. Each is of arbitrary length. In particular, floating-point numbers can be of any length. Picture and word data are of arbitrary format and length.

A complete Connection Machine memory address has three parts. The first part indicates a physical processor. The second part indicates one of the virtual processors simulated by that physical processor. (This part is empty if there is only one virtual processor per physical processor.) The third part is an address within the memory of that virtual processor.

Data may be exchanged between the Connection Machine memory and the front end in any of three ways: slicewise, processorwise, and arraywise. *Read-slice* reads a single bit of information from the memory of each of a series of consecutive processors, assembles them into a signed integer, and passes the integer to the front end. *Write-slice* moves data from the front end to the Connection Machine memory. Slice operations are typically done 16 or 32 processors at a time. *Read-processor* and *write-processor* move a single field between the front end and a single processor. *Read-array* and *write-array* move arrays of fields between the front end and a set of contiguous processors.

7.2 Connection Machine Instruction Flow

All instructions flow into the Connection Machine hardware from the front end. These macro-instructions are sent to a microcontroller, which expands them into a series of nano-instructions. Some expand into just a few nano-instructions. Others expand into hundreds or thousands. It is also possible to feed nano-level instructions to the microcontroller and control the hardware directly. It is not, however, efficient to do so, because the front-end cannot supply these instructions rapidly enough to keep the system busy. (Direct control of the hardware from the front end is provided primarily so that the front end can support debugging and diagnostic aids.)

Nano-instructions are broadcast to all processors in parallel. Processors, however, have the option of "sitting out" a series of instructions. A one-bit flag within each processor, the *context flag*, determines whether that individual processor will respond to the instruction

or not. Most of the instructions discussed in this chapter are "conditional" in the sense that they take effect only in the processors that are *active*, that is, whose *context flag* is 1.

The Connection Machine system is implemented with four physical microcontrollers, one for each section of 16,384 processors. If the system has a single front end, that front end is connected to all four microcontrollers and therefore drives all 65,536 processors. A system may be configured with up to four front ends. A crossbar switch called the Nexus makes the connections between front ends and microcontrollers. It is possible, therefore, to have four users operating simultaneously. Each works at a separate front end, and each has a separate instruction stream executing in a section of the system's processors. The examples in this chapter, however, assume that the system is operating with a single front end.

7.3 Computational and Global Instructions

Computational instructions operate on signed integers, unsigned integers, and floating-point values. They include unary operators such as ***not***, ***negate***, ***absolute value***, and *square root*. All standard binary operators such as ***add***, ***subtract***, ***multiply***, ***divide***, ***compare***, and ***shift*** are included. These instructions operate in all processors simultaneously; each processor uses the data that is stored in that processor's memory.

The *random* instruction places an independently chosen pseudo-random number in each processor. Two processors may or may not be assigned the same random value.

Global instructions produce a single result from data items stored in the memories of all selected processors. *Global-logior*, for example, takes the inclusive OR of a field in each processor's memory. *Global-count* examines a single-bit field in all processors and returns the number of "1" bits. *Global-add* sums multi-bit fields. *Global-max* and *global-min* return the largest (smallest) value found in a specified field across all selected processors. *Global-add* operates on unsigned integers, signed integers, or floating point values, as do *global-max* and *global-min*. The *enumerate* instruction places a different consecutive integer into each of a selected set of processors.

7.4 Communications Instructions

The simplest form of communication between Connection Machine processors is between nearest neighbors. Each processor is wired to its neighbors to the North, East, West, and South by a communications network called the *NEWS grid*. Four instructions, *get-from-north*, *get-from-east*, *get-from-west*, and *get-from-south* control the transfer of data. Information is passed one bit at a time.

General intercommunication and dynamic reconfiguration is performed by a much more

powerful communications system, the Connection Machine *router*. It allows full messages to be sent from any processor to any other; the sending processor simply needs to have the address of the destination processor. Messages may be of any length. Typical messages contain 32 bits of information; adding the address information and headers results in a transmitted package of 50 to 60 bits (depending on the number of virtual processors being used).

Each of the 65,536 physical processors is connected to 16 other physical processors in a special organization (a 16-dimensional hypercube) that provides large numbers of direct paths to distant parts of the system. Every processor is connected to 16 other processors, namely those whose binary address is different in just one of the 16 bits. The following example shows the interconnections of processors 6_{10} and 2070_{10}. The binary addresses are shown in parentheses.

```
    2 ( 0000  0000  0000  0010 )
    4 ( 0000  0000  0000  0100 )
    6 ( 0000  0000  0000  0110 )
    7 ( 0000  0000  0000  0111 )
   14 ( 0000  0000  0000  1110 )
   22 ( 0000  0000  0001  0110 )
   38 ( 0000  0000  0010  0110 )
   70 ( 0000  0000  0100  0110 )
  134 ( 0000  0000  1000  0110 )
  262 ( 0000  0001  0000  0110 )
  518 ( 0000  0010  0000  0110 )
 1030 ( 0000  0100  0000  0110 )
 2054 ( 0000  1000  0000  0110 )
 4102 ( 0001  0000  0000  0110 )
 8198 ( 0010  0000  0000  0110 )
16390 ( 0100  0000  0000  0110 )
32774 ( 1000  0000  0000  0110 )


   22 ( 0000  0000  0001  0110 )
 2054 ( 0000  1000  0000  0110 )
 2066 ( 0000  1000  0001  0010 )
 2068 ( 0000  1000  0001  0100 )
 2070 ( 0000  1000  0001  0110 )
 2071 ( 0000  1000  0001  0111 )
 2078 ( 0000  1000  0001  1110 )
```

```
 2102 ( 0000  1000  0011  0110 )
 2134 ( 0000  1000  0101  0110 )
 2198 ( 0000  1000  1001  0110 )
 2326 ( 0000  1001  0001  0110 )
 2582 ( 0000  1010  0001  0110 )
 3094 ( 0000  1100  0001  0110 )
 6166 ( 0001  1000  0001  0110 )
10262 ( 0010  1000  0001  0110 )
18454 ( 0100  1000  0001  0110 )
34838 ( 1000  1000  0001  0110 )
```

These two sets of addresses have a common connection. Processors 6 and 2070 both connect to 22. Thus it is possible to pass a message, for example, from processor 14 to processor 10262 in just four steps. The router at processor 14 passes it to the router at processor 6, which passes it to 22. From there it goes to 2070 and then to 10262.

7.5 The Routing Process

Connection Machine physical processors are grouped sixteen to a chip. There is a single router on each chip that services all sixteen processors. Hence four of the sixteen routing connections are internal to an individual chip. It takes a maximum of twelve steps to move from any chip to any other chip. During message routing, the system goes through all twelve steps. If the router on a given chip has a message whose relative address has a "1" in the low order bit position, it sends that message on the first of the twelve steps to the chip whose address differs in that same bit (i.e., the next chip). If the message it has has a "0" in the low order relative address bit, the on-chip router does not send any data on that step. The process continues through all twelve steps, with all router chips responding in the same way.

The basic message passing instruction is *send*. Arguments to *send* specify the length of the message and two memory fields. Within each processor, one field contains the message data, and the other contains the address of a destination processor. *Send* causes all active processors to initiate message transfers at once. The special Connection Machine routing hardware handles the volume of messages efficiently. An individual router on a chip may receive as many as twelve messages from other chips during a message cycle, one from each other chip that it is connected to. It can in turn send as many as twelve messages, one on each of the wires. If two messages need to go down the same wire, one is buffered until the next routing cycle. If an individual router becomes extremely busy, it can defer acceptance of any new messages from its own processors. Deferral keeps the router free to

handle messages from other chips. If the chip's buffer space still fills, it refers messages to neighboring chips.

Simultaneous message sending introduces the possibility that the same location in the same processor will receive two or more messages in the same cycle. The simple *send* instruction gives unpredictable results in this case. Several variations of the *send* instruction, such as *send-with-add*, deal with this possibility. If two or more *send-with-add* messages arrive at the same destination, they are summed. *Send-with-overwrite* causes one message to be delivered intact, discarding all other messages directed to that destination. Other options include *send-with-max* and *send-with-logior*.

7.6 Dynamic Reconfiguration

A processor address is all it takes to establish a link on the system. This flexibility allows applications to reconfigure dynamically. A number of intructions support this capability. The *my-address* instruction allows processors to determine their own addresses, so they can send them to other processors and thus establish new connections. The *processor-cons* instruction allows each selected processor to find another "free" processor.

Processor-cons specifies the address of a one-bit field, the "free flag." A processor is considered free if it has a "1" in that field. The system looks in parallel for processors with 1's and passes to each selected processor the address of a different free processor, and at the same time clears the free flags of those free processors.

Chapter 8

Looking to the Future

At one level this report is about algorithms for data level parallel computers: algorithms for looking at the whole problem at once. But at a deeper and more important level, it is really the story of what happened when three very creative people teamed up with a new style of computer, the Connection Machine system. All three people saw new ways to break through old barriers. The computer allowed them to confirm their intuition quickly and then to build upon that intuition.

The intuitive insight behind the document retrieval algorithm is the fact that documents contain a rich set of synonyms for their main content topics. Comparing whole documents could eliminate the need to play guessing games with key words. The idea had never been effectively tested because no conventional computer could execute the algorithms quickly on large data bases. In fact, the first tests on document retrieval by whole document comparison were not particularly encouraging. They were run on a data base of 150 documents, which turned out to be inadequate. When the test was widened to 1500 documents, results were more encouraging. At the level of 15,000 documents, they were outstanding. Without a data level parallel computer such as the Connection Machine system, there would have been no way to even try the approach with 15,000 documents. Test runs would have taken days. Interaction would have been impossible. Now that it has been shown that the algorithm works, whole new possibilities for data base system design are opening up.

The intuitive insight behind the fluid flow algorithm is the fact the behavior of fluids can be simulated without extensive arithmetic computations. Modeling the primitive behavior of molecule packets on a large enough scale can elicit the same macroscopic behavior as real fluids. Tests on the Connection Machine computer suggest strongly that it does. The result is a new and potentially important avenue of scientific investigation.

The intuitive insight behind the contour mapping algorithm is the fact that sophisticated image processing and vision algorithms can be tested on large amounts of data with a small amount of programming effort. The drawing of contour maps, for example, is greatly

simplified by data level parallelism, because it is not necessary to identify the contours one by one and then traverse the perimeter of each one sequentially; instead, each pixel of the contour map "draws itself" in parallel with all the other pixels. Instead of having to break up each phase of the problem into smaller pieces for sequencing purposes, the programmer can tackle it all at once. The result is smaller and simpler programs.

The revolution in data level parallel computing is here. The three algorithms described in this report are only a beginning. But they make an important point: innovative users are an integral part of the story. Users who are stimulated to look at old problems in new ways. Users who revisit problems given up on as impossible in the 60's and 70's. Users who know that a simpler solution is a better solution. These are the users who will assure that the future belongs to computers that look at the whole problem at once.

Bibliography

[1] David C. Blair and M. E. Maron. An evaluation of retrieval effectiveness for a full-text document-retrieval system. *Comm. ACM*, 28(3):289–267, March 1985.

[2] John F. Canny. *Finding Lines and Edges in Images.* AI Memo 720, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, 1983.

[3] Michael Drumheller and Tomaso Poggio. On parallel stereo. In *International Conference on Robotics and Automation*, IEEE, April 1986.

[4] U. Frisch, B. Hasslacher, and Y. Pomeau. *A Lattice Gas Automaton for the Navier-Stokes equation.* Preprint LA-UR-85-3503, Los Alamos, 1985.

[5] W. Eric L. Grimson. *From Images to Surface.* MIT Press, Cambridge, Massachusetts, 1981.

[6] Samuel P. Harbison and Guy L. Steele Jr. *C: A Reference Manual.* Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[7] J. Hardy, O. de Pazzis, and Y. Pomeau. Molecular dynamics of a classical lattice gas: transport properties and time correlation functions. *Phys. Rev.*, A13(1949), 1976.

[8] W. Daniel Hillis. *The Connection Machine.* MIT Press, Cambridge, Massachusetts, 1985.

[9] Guy L. Steele Jr., Scott E. Fahlman, Richard P. Gabriel, David A. Moon, and Daniel L. Weinreb. *Common Lisp: The Language.* Digital Press, Burlington, Massachusetts, 1984.

[10] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[11] David Marr. *Vision.* W. H. Freeman, San Francisco, 1982.

[12] David Marr and Ellen Hildreth. Theory of edge detection. *Proc. Roy. Soc. London*, B(207):187–217, 1980.

[13] K. Prazdny. Detection of binocular disparities. *Biological Cybernetics*, 52:93–99, 1985.

[14] James B. Salem and Stephen Wolfram. *Thermodynamics and Hydrodynamics with Cellular Automata.* Internal technical report, Thinking Machines Corporation, Cambridge, Massachusetts, November 1985.

[15] David S. Touretzky. *Lisp: A Gentle Introduction to Symbolic Computation.* Harper & Row, New York, 1984.

[16] Patrick Henry Winston and Berthold Klaus Paul Horn. *Lisp.* Addison-Wesley, Reading, Massachusetts, second edition, 1984.